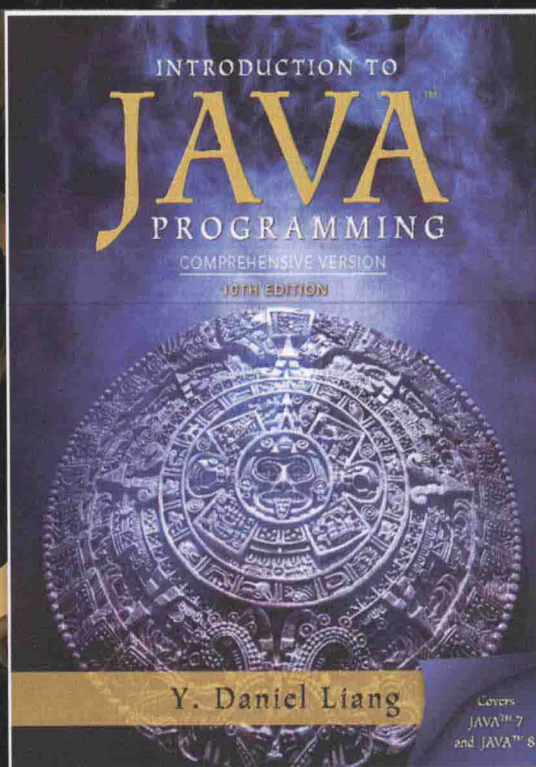


Java语言程序设计 (进阶篇)

[美] 梁勇 (Y. Daniel Liang) 著 戴开宇 译
阿姆斯特朗亚特兰大州立大学 复旦大学

Introduction to Java Programming
Comprehensive Version Tenth Edition



计 算 机 科 学 丛 书

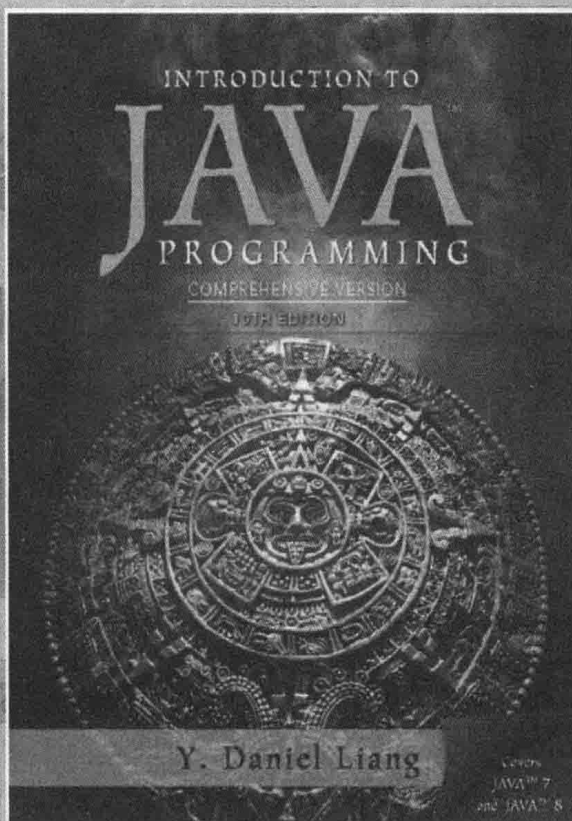
原书第10版

Java语言程序设计

(进阶篇)

[美] 梁勇 (Y. Daniel Liang) 著 戴开宇 译
阿姆斯特朗亚特兰大州立大学 复旦大学

Introduction to Java Programming
Comprehensive Version Tenth Edition



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

Java 语言程序设计 (进阶篇) (原书第 10 版) / (美) 梁勇 (Y. Daniel Liang) 著 ; 戴开宇译 .
—北京 : 机械工业出版社 , 2016.9

(计算机科学丛书)

书名原文 : Introduction to Java Programming, Comprehensive Version, Tenth Edition

ISBN 978-7-111-54856-0

I. J… II. ① 梁… ② 戴… III. JAVA 语言 - 程序设计 IV. TP312.8

中国版本图书馆 CIP 数据核字 (2016) 第 222315 号

本书版权登记号 : 图字 : 01-2014-5467

Authorized translation from the English language edition, entitled Introduction to Java Programming, Comprehensive Version, Tenth Edition, 978-0-13-376131-3 by Y. Daniel Liang, published by Pearson Education, Inc., Copyright © 2015, 2013, 2011.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2016.

本书中文简体字版由 Pearson Education (培生教育出版集团) 授权机械工业出版社在中华人民共和国境内 (不包括香港、澳门特别行政区及台湾地区) 独家出版发行。未经出版者书面许可 , 不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education (培生教育出版集团) 激光防伪标签 , 无标签者不得销售。

本书是 Java 语言的经典教材 , 中文版分为基础篇和进阶篇 , 主要介绍程序设计基础、面向对象程序设计、GUI 程序设计、数据结构和算法、高级 Java 程序设计等内容。本书以示例讲解解决问题的技巧 , 提供大量的程序清单 , 每章配有复习题和编程练习题 , 帮助读者掌握编程技术 , 并应用所学技术解决实际应用开发中遇到的问题。

进阶篇主要介绍线性表、栈、队列、集合、映射表、排序、二叉查找树、AVL 树、散列、图及其应用、并行程序设计、网络、Java 数据库程序设计以及 JSF 等内容。

本书可作为高等院校相关专业程序设计课程的基础教材 , 也可作为 Java 语言及编程爱好者的参考资料。

出版发行 : 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑 : 曲 熠

责任校对 : 董纪丽

印 刷 : 北京市荣盛彩色印刷有限公司

版 次 : 2016 年 10 月第 1 版第 1 次印刷

开 本 : 185mm × 260mm 1/16

印 张 : 30.5

书 号 : ISBN 978-7-111-54856-0

定 价 : 89.00 元

凡购本书 , 如有缺页、倒页、脱页 , 由本社发行部调换

客服热线 : (010) 88378991 88361066

投稿热线 : (010) 88379604

购书热线 : (010) 68326294 88379649 68995259

读者信箱 : hzjsj@hzbook.com

版权所有 · 侵权必究

封底无防伪标均为盗版

本书法律顾问 : 北京大成律师事务所 韩光 / 邹晓东

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域中取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅肇划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson, McGraw-Hill, Elsevier, MIT, John Wiley & Sons, Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum, Bjarne Stroustrup, Brian W. Kernighan, Dennis Ritchie, Jim Gray, Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, Abraham Silberschatz, William Stallings, Donald E. Knuth, John L. Hennessy, Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方式如下：

华章网站：www.hzbook.com

电子邮件：hzjsj@hzbook.com

联系电话：(010) 88379604

联系地址：北京市西城区百万庄南街1号

邮政编码：100037



华章科技图书出版中心

中文版序

Introduction to Java Programming, Comprehensive Version, Tenth Edition

Welcome to the Chinese translation of Introduction to Java Programming Tenth Edition. The first edition of the English version was published in 1998. Since then ten editions of the book have been published in the last seventeen years. Each new edition substantially improved the book in contents, presentation, organization, examples, and exercises. This book is now the #1 selling computer science textbook in the US. Hundreds and thousands of students around the world have learned programming and problem solving using this book.

I thank Dr. Kaiyu Dai of Fudan University for translating this latest edition. It is a great honor to reconnect with Fudan through this book. I personally benefited from teachings of many great professors at Fudan. Professor Meng Bin made Calculus easy with many insightful examples. Professor Liu Guangqi introduced multidimensional mathematic modeling in the Linear Algebra class. Professor Zhang Aizhu laid a solid mathematical foundation for computer science in the discrete mathematics class. Professor Xia Kuanli paid a great attention to small details in the PASCAL course. Professor Shi Bole showed many interesting sort algorithms in the data structures course. Professor Zhu Hong required an English text for the algorithm design and analysis course. Professor Lou Rongsheng taught the database course and later supervised my master's thesis.

My study at Fudan and teaching in the US prepared me to write the textbook. The Chinese teaching emphasizes on the fundamental concepts and basic skills, which is exactly I used to write this book. The book is fundamentals first by introducing basic programming concepts and techniques before designing custom classes. The fundamental-first approach is now widely adopted by the universities in the US. With the excellent translation from Dr. Dai, I hope more students will benefit from this book and excel in programming and problem solving.

Y. Daniel Liang

欢迎阅读本书第 10 版的中文版。本书英文版的第 1 版于 1998 年出版。自那之后的 17 年中，本书共出版了 10 个版本。每个新的版本都在内容、表述、组织、示例以及练习题等方面进行了大量的改进。本书目前在美国计算机科学类教材中销量排名第一。全世界无数的学生通过本书学习程序设计以及问题求解。

感谢复旦大学的戴开宇博士翻译了这一最新版本。非常荣幸通过这本书和复旦大学重建联系，我本人曾经受益于复旦大学的许多杰出教授：孟斌教授采用许多富有洞察力的示例将微积分变得清晰易懂；刘光奇教授在线性代数课堂上介绍了多维度数学建模；张霭珠教授的离散数学课程为计算机科学的学习打下了坚实的数学基础；夏宽理教授在 Pascal 课程中对许多小的细节给予了极大的关注；施伯乐教授在数据结构课程中演示了许多有趣的排序算法；朱洪教授在算法设计和分析课程中使用了英文教材；楼荣生教授讲授了数据库课程，并且指导了我的硕士论文。

我在复旦大学的学习经历以及美国的授课经验为撰写本书奠定了基础。中国的教学重视基本概念和基础技能，这也是我写这本书所采用的方法。本书采用基础为先的方法，在介绍设计自定义类之前首先介绍了基本的程序设计概念和方法。目前，基础为先的方法也被美国的大学广泛采用。我希望通过戴博士的优秀翻译，让更多的学生从中受益，并在程序设计和问题求解方面出类拔萃。

梁勇

y.daniel.liang@gmail.com

www.cs.armstrong.edu/liang

Java 是一门伟大的程序设计语言，同时，它还是基于 Java 语言从嵌入式开发到企业级开发的平台。在风起云涌的计算机技术发展历程中，Java 的身影随处可见，而且生命力极其强大。1995 年，Java Applet 使得 Web 网页可以表现精彩和互动的多媒体内容，促进了 Web 的蓬勃发展。之后随着 Web 的发展，应用 Web 成为大型应用开发的主流方式，Java 凭借其“一次编译，到处运行”的特性很好地支持了互联网应用所要求的跨平台能力，成为服务器端开发的主流语言。Java EE 至今依然是最重要的企业开发服务器端平台。2004 年再次产生了对 Web 客户端体验的强烈需求，促使富因特网应用技术广泛流行，从 Java Web Start 到现在的 JavaFX，都是重要的富因特网应用技术。现在我们进入了移动互联网时代，而 Java 依然是当之无愧的主角。从第一阶段移动互联网中的 J2ME，到目前移动操作系统中全球占据份额最大的 Android 系统上的 App 开发，都采用的是 Java 语言和平台。云计算、大数据、物联网、可穿戴设备等技术的应用，都需要可以跨平台、跨设备的分布式计算环境，我们依然会看到 Java 语言在其中的关键作用。除此之外，Java 还是一门非常优秀的教学语言。它是一门经典的面向对象编程语言，拥有优雅和尽量简明的语法以及丰富的实用类库，让编程人员可以尽可能地将精力集中在业务领域的问题求解上。许多开源项目和科研中的原型系统都是采用 Java 实现的。课堂教学采用的语言同时在工业界和学术领域具有如此广泛的应用，对于学生今后的科研和工作都有直接帮助。我曾经对美国计算机专业排名靠前的几十所大学的相关课程进行调研，这些著名大学的编程课程中绝大部分选用了 Java 语言进行教学。

在多年前机械工业出版社举办的一次教学研讨会上，我有幸认识了原书的作者梁勇（Y. Daniel Liang）教授并进行了交流。那次会议之后我开始在主讲的程序设计课程中采用本书英文版作为教材，在同行和学生中得到了良好反响。作为复旦校友，梁教授对中国学生的情况非常了解，书中没有过于晦涩的词汇和表达，所以本英文教材非常适合中国学生的英文基础。更重要的是，本书知识点全面，体系结构清晰，重点突出，文字准确，内容组织循序渐进，并有大量精选的示例和配套素材，比如精心设计的大量练习题，甚至在配套网站中有支持教学的大量动画演示。本书采用基础优先的方式，从编程基础开始，逐步引入面向对象思想，最后介绍应用框架，这样很适合程序设计入门的学生。另外，强调面向问题求解的教学方法是本书特色，这也是我在课堂上一一直遵循的教学方法。通过生动实用的例子来引导学生学习程序设计课程，避免了枯燥的语法学习，让学生学以致用，并且可以举一反三。程序设计课堂最重要的是要培养学生的计算思维，这对学生综合素质的培养以及其他知识的学习都是很有裨益的。掌握了程序设计的思维，可以很方便地学习和使用其他编程语言。该版本的另一特色是对最新 Java 语言特色的跟进，即基于 Java 最新版本 8 进行介绍。这是 Java 语言变动非常大的一个版本，比如对 JavaFX 的全面引入以及并行计算的支持等，都反映了最新的计算机技术和应用特点。相应地，教材也进行了大幅更新。我很荣幸成为本书第 10 版的译者，让中国的读者可以通过这一最新版本的中文版方便地学习程序设计相关知识。

在本书的翻译过程中，我得到了原书作者梁勇教授的大力支持。非常感谢他不仅对我邮件中的一些问题进行快速回复和详细解答，还拨冗写了中文版序，其一丝不苟的学术精神让人感动。感谢机械工业出版社的朱劼编辑，她在本书的整个翻译过程中提供了许多帮助。感谢所有为本书付出心血的出版社工作人员以及本书前一版的译者，本书的出版也得益于他们的工作。最后要感谢我的家人在翻译过程中给予的支持和鼓励。由于经验不足和水平有限，书中难免会存在问题，敬请大家指正。你们善意的指正，对我和阅读本书的许多读者是有益的。

戴开宇

2016年7月

许多读者就本书之前的版本给出了很多反馈。这些评论和建议极大地改进了本书。这一版在表述、组织、示例、练习题以及附录方面都进行了极大的增强，包括：

- 用 JavaFX 取代了 Swing。JavaFX 是一个用于开发 Java GUI 程序的新框架，它极大地简化了 GUI 程序设计，比 Swing 更易于学习。
- 在 GUI 程序设计之前介绍异常处理、抽象类和接口，若教师选择不教授 GUI 的内容，可以直接跳过第 14 ~ 16 章。
- 在第 4 章便开始介绍对象和字符串，从而使得学生可以较早地使用对象和字符串来开发有趣的程序。
- 包含更多新的有趣示例和练习题，用于激发学生兴趣。在配套网站 (www.cs.armstrong.edu/liang/intro10e/ 或 www.pearsonhighered.com/liang) 上还为教师提供了 100 多道编程练习题。

本书采用基础优先的方法，在设计自定义类之前，首先介绍基本的程序设计概念和技术。选择语句、循环、方法和数组这样的基本概念和技术是程序设计的基础，它们为学生进一步学习面向对象程序设计和高级 Java 程序设计做好准备。

本书以问题驱动的方式来教授程序设计，将重点放在问题的解决而不是语法上。我们通过使用在各种应用情景中引发思考的问题，使得程序设计的介绍也变得更加有趣。前面章节的主线放在问题的解决上，引入合适的语法和库以支持编写解决问题的程序。为了支持以问题驱动的方式来教授程序设计，本书提供了大量不同难度的问题来激发学生的积极性。为了吸引各个专业的学生来学习，这些问题涉及很多应用领域，包括数学、科学、商业、金融、游戏、动画以及多媒体等。

本书将程序设计、数据结构和算法无缝集成在一起，采用一种实用性的方式来教授数据结构。首先介绍如何使用各种数据结构来开发高效的算法，然后演示如何实现这些数据结构。通过实现，学生获得关于数据结构效率，以及如何和何时使用某种数据结构的深入理解。最后，我们设计和实现了针对树和图的自定义数据结构。

本书广泛应用于全球各大学的程序设计入门、数据结构和算法课程中。完全版包括程序设计基础、面向对象程序设计、GUI 程序设计、数据结构、算法、并行、网络、数据库和 Web 程序设计。这个版本旨在把学生培养成精通 Java 的程序员。基础篇可用于程序设计的第一门课程（通常称为 CS1）。基础篇包含完全版的前 18 章内容，前 13 章适合准备 AP 计算机科学考试（AP Computer Science Exam）的人员使用。

教授编程的最好途径是通过示例，而学习编程的唯一途径是通过动手练习。本书通过示例对基本概念进行了解释，提供了大量不同难度的练习题供学生进行实践。在我们的程序设

⊖ 本书中文版将完全版分为基础篇和进阶篇出版，基础篇对应原书第 1 ~ 18 章，进阶篇对应原书第 19 ~ 33 章，您手中的这一本是进阶篇。——编辑注

⊖ 关于本书配套资源，用书学生和教师可向培生教育出版集团北京代表处申请，电话：010-5735 5169/5735 5171，电子邮件：service.cn@pearson.com。——编辑注

计课程中，每次课后都布置了编程练习。

我们的目标是编写一本可以通过各种应用场景中的有趣示例来教授问题求解和程序设计的教材。如果您有任何关于如何改进本书的评论或建议，请通过以下方式与我联系。

Y. Daniel Liang

y.daniel.liang@gmail.com

www.cs.armstrong.edu/liang

www.pearsonhighered.com/liang

本版新增内容

本版对各个细节都进行了全面修订，以增强其清晰性、表述、内容、例子和练习题。本版主要的改进如下：

- 更新到 Java 8 版本。
- 由于 Swing 被 JavaFX 所替代，因此所有的 GUI 示例和练习题都使用 JavaFX 改写。
- 使用 lambda 表达式来简化 JavaFX 和线程中的编程。
- 在配套网站上为教师提供了 100 多道编程练习题，并给出了答案。这些练习题没有出现在教材中。
- 在第 4 章就引入了数学方法，使得学生可以使用数学函数编写代码。
- 在第 4 章就引入了字符串，使得学生可以早点使用对象和字符串开发有趣的程序。
- GUI 编程放在抽象类和接口之后介绍，若教师选择不教授 GUI 内容的话，可以直接跳过这些章节。
- 第 4、14、15 和 16 章是全新的章节。
- 第 28 和 29 章大幅改写，对最小生成树和最短路径使用更加简化的方法实现。

教学特色

本书使用以下要素组织素材：

- **教学目标** 在每章开始处列出学生应该掌握的内容，学完这章后，学生能够判断自己是否达到这个目标。
- **引言** 提出代表性的问题，以便学生对该章内容有一个概括了解。
- **要点提示** 突出每节中涵盖的重要概念。
- **复习题** 按节组织，帮助学生复习相关内容并评估掌握的程度。
- **示例学习** 通过精心挑选示例，以容易理解的方式教授问题求解和程序设计概念。本书使用多个小的、简单的、激发兴趣的例子来演示重要的概念。
- **本章小结** 回顾学生应该理解和记住的重要主题，有助于巩固该章所学的关键概念。
- **测试题** 测试题是在线的，让学生对编程概念和技术进行自我测试。
- **编程练习题** 为学生提供独立应用所学新技能的机会。练习题的难度分为容易（没有星号）、适中（*）、难（**）和具有挑战性（***）四个级别。学习程序设计的窍门就是实践、实践、再实践。所以，本书提供了大量的编程练习题。
- **注意、提示、警告和设计指南** 贯穿全书，对程序开发的重要方面提供有价值的建议和见解。
 - **注意** 提供学习主题的附加信息，巩固重要概念。

- **提示** 教授良好的程序设计风格和实践经验。
- **警告** 帮助学生避开程序设计错误的误区。
- **设计指南** 提供设计程序的指南。

灵活的章节顺序

本书提供灵活的章节顺序，使学生可以或早或晚地了解 GUI、异常处理、递归、泛型和 Java 集合框架等内容。下页的插图显示了各章之间的相关性。

本书的组织

所有的章节分为五部分，构成 Java 程序设计、数据结构和算法、数据库和 Web 程序设计的全面介绍。因为知识是循序渐进的，前面的章节介绍了程序设计的基本概念，并且通过简单的例子和练习题指导学生；后续的章节逐步详细地介绍 Java 程序设计，最后介绍开发综合的 Java 应用程序。附录包含各种主题，包含数系、位操作、正则表达式以及枚举类型。

第一部分 程序设计基础（第 1～8 章）

本书第一部分是基石，让你开始踏上 Java 学习之旅。你将开始了解 Java（第 1 章），还将学习像基本数据类型、变量、常量、赋值、表达式以及操作符这样的基本程序设计技术（第 2 章），选择语句（第 3 章），数学函数、字符和字符串（第 4 章），循环（第 5 章），方法（第 6 章），数组（第 7～8 章）。在第 7 章之后，可以跳到第 18 章去学习如何编写递归的方法来解决本身具有递归特性的问题。

第二部分 面向对象程序设计（第 9～13 章和第 17 章）

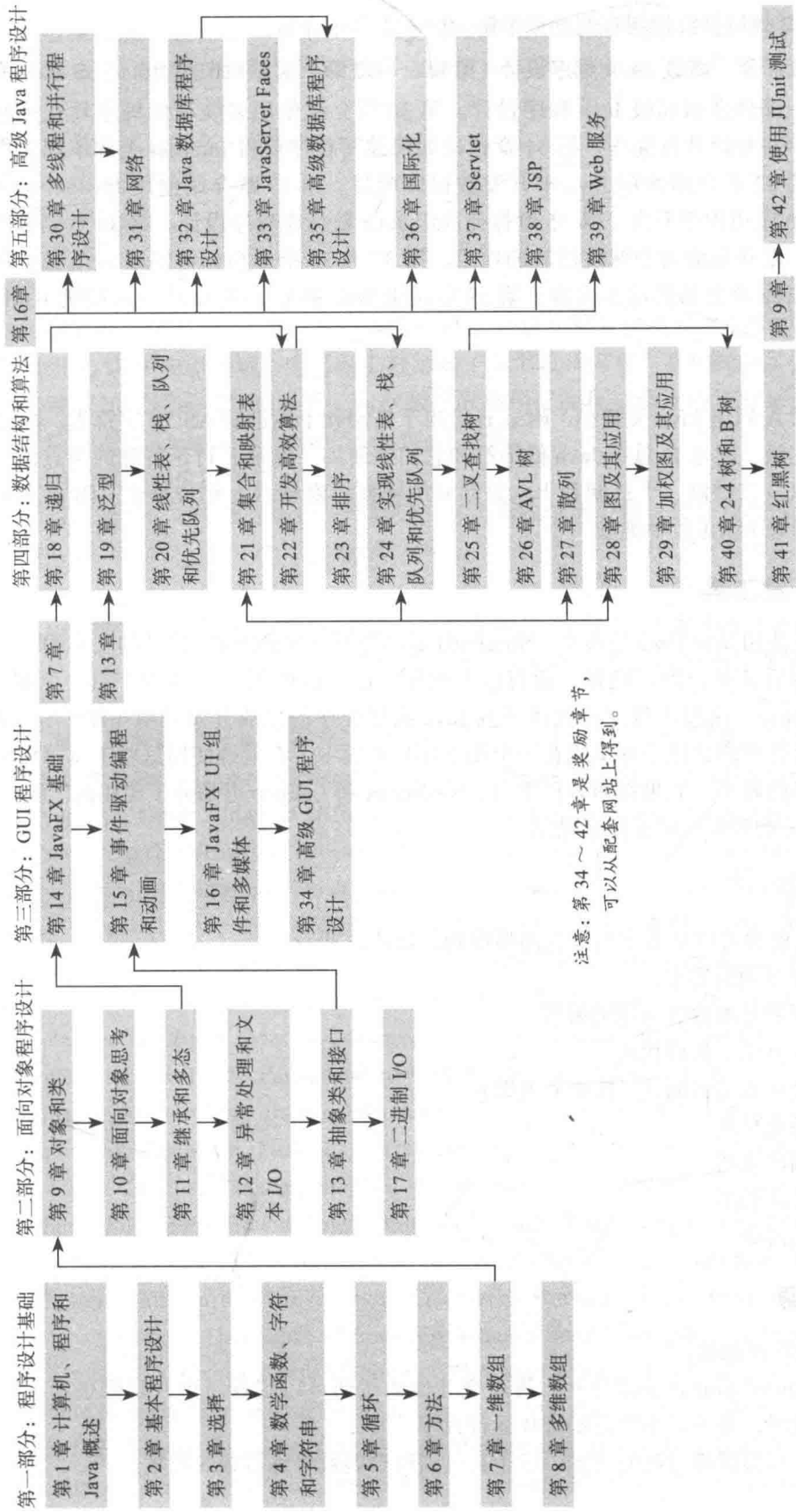
这一部分介绍面向对象程序设计。Java 是一种面向对象程序设计语言，它使用抽象、封装、继承和多态来提供开发软件的极大灵活性、模块化和可重用性。你将学习如何使用对象和类进行程序设计（第 9～10 章）、类的继承（第 11 章）、多态性（第 11 章）、异常处理（第 12 章）、抽象类（第 13 章）以及接口（第 13 章）。文本 I/O 将在第 12 章介绍，二进制 I/O 将在第 17 章介绍。

第三部分 GUI 程序设计（第 14～16 章和奖励章节第 34 章）

JavaFX 是一个开发 Java GUI 程序的新框架。它不仅对于开发 GUI 程序有用，还是一个用于学习面向对象程序设计的优秀教学工具。这一部分中在第 14～16 章介绍使用 JavaFX 的 Java GUI 程序设计。主要的主题包括 GUI 基础（第 14 章）、容器面板（第 14 章）、绘制形状（第 14 章）、事件驱动编程（第 15 章）、动画（第 15 章）、GUI 组件（第 16 章），以及播放音频和视频（第 16 章）。你将学习采用 JavaFX 的 GUI 程序设计的架构，并且使用组件、形状、面板、图像和视频来开发有用的应用程序。第 34 章涵盖 JavaFX 的高级特性。

第四部分 数据结构和算法（第 18～29 章和奖励章节第 40～41 章）

这一部分介绍经典数据结构和算法课程中的主要内容。第 18 章介绍递归来编写解决本身具有递归特性的问题的方法。第 19 章介绍泛型来提高软件的可靠性。第 20 和 21 章介绍 Java 合集框架，它为数据结构定义了一套有用的 API。第 22 章讨论算法效率的度量以便给应用程序选择合适的算法。第 23 章介绍经典的排序算法。你将在第 24 章中学到如何实现经典的数据结构，如线性表、队列和优先队列。第 25 和 26 章介绍二叉查找树和 AVL 树。第 27 章介绍散列以及通过散列实现映射表（map）和集合（set）。第 28 和 29 章介绍图的应用。



2-4 树、B 树以及红黑树在奖励章节第 40 ~ 41 章中介绍。

第五部分 高级 Java 程序设计 (第 30 ~ 33 章、奖励章节第 35 ~ 39 章及第 42 章)

这一部分介绍高级 Java 程序设计。第 30 章介绍使用多线程使程序具有更好的响应和交互性,并介绍并行编程。第 31 章讨论如何编写程序使得 Internet 上的不同主机能够相互对话。第 32 章介绍使用 Java 来开发数据库项目。第 33 章介绍使用 JavaServer Faces 进行现代 Web 应用程序开发。第 35 章探究高级 Java 数据库程序设计。第 36 章涵盖国际化支持的使用,以开发面向全球使用者的项目。第 37 和 38 章介绍如何使用 Java servlet 和 JSP 创建来自 Web 服务器的动态内容。第 39 章讨论 Web 服务。第 42 章介绍使用 JUnit 测试 Java 程序。

附录

附录 A 列出 Java 关键字。附录 B 给出十进制和十六进制 ASCII 字符集。附录 C 给出操作符优先级。附录 D 总结 Java 修饰符和它们的使用。附录 E 讨论特殊的浮点值。附录 F 介绍数系以及二进制、十进制和十六进制间的转换。附录 G 介绍位操作。附录 H 介绍正则表达式。附录 I 涵盖枚举类型。

Java 开发工具

可以使用 Windows 记事本 (NotePad) 或写字板 (WordPad) 这样的文本编辑器创建 Java 程序,然后从命令窗口编译、运行这个程序。也可以使用 Java 开发工具,例如,NetBeans 或者 Eclipse。这些工具支持快速开发 Java 应用程序的集成开发环境 (IDE),编辑、编译、构建、运行和调试程序都集成在一个图形用户界面中。有效地使用这些工具可以极大地提高编写程序的效率。如果按照教程学习,NetBeans 和 Eclipse 也是易于使用的。关于 NetBeans 和 Eclipse 的教程,参见配套网站。

学生资源

学生资源可以从本书的配套网站得到,具体包括:

- 复习题的答案。
- 偶数号编程练习题的解答。
- 本书例子的源代码。
- 交互式的自测题 (按章节组织)。
- 补充材料。
- 调试技巧。
- 算法动画。
- 勘误表。

教师资源

教师资源包括:

- PowerPoint 教学幻灯片,通过交互性的按钮可以观看彩色并且语法项高亮显示的源代码,并可以不离幻灯片运行程序。
- 所有编程练习题的答案。学生只可以得到偶数号练习题的答案。

- 100 多道编程练习题，按章节组织。这些练习题仅对教师开放，并提供答案。
- 基于 Web 的测试题生成器。(教师可以选择章节以从 2000 多个大型题库中生成测试题。)
- 样卷。大多数试卷包含 4 个部分：
 - 多选题或者简答题。
 - 改正编程错误。
 - 跟踪程序。
 - 编写程序。
- ACM/IEEE 课程体系 2013 版。新的 ACM/IEEE 计算机科学课程体系 2013 版将知识主体组织成 18 个知识领域。为了帮助教师基于本书设计课程，我们提供了示例教学大纲来确定知识领域和知识单元。示例教学大纲用于一个三学期的课程系列，作为一个学院自定义 (institutional customization) 示例。
- 具有 ABET 课程评价的样卷。
- 课程项目。通常，每个项目给出一个描述，并且要求学生分析、设计和实现该项目。

致谢

感谢阿姆斯特兰亚特兰大州立大学给我机会讲授我所写的内容，并支持我将所教的内容编写成教材。教学是我持续改进本书的灵感之源。感谢使用本书的教师和学生提出的评价、建议、错误报告和赞扬。

由于有了对本版和以前版本的富有见解的审阅，本书得到很大的改进。感谢以下审阅人员：Elizabeth Adams (James Madison University), Syed Ahmed (North Georgia College and State University), Omar Aldawud (Illinois Institute of Technology), Stefan Andrei (Lamar University), Yang Ang (University of Wollongong, Australia), Kevin Bierre (Rochester Institute of Technology), David Champion (DeVry Institute), James Chegwiddden (Tarrant County College), Anup Dargar (University of North Dakota), Charles Dierbach (Towson University), Frank Ducrest (University of Louisiana at Lafayette), Erica Eddy (University of Wisconsin at Parkside), Deena Engel (New York University), Henry A Etlinger (Rochester Institute of Technology), James Ten Eyck (Marist College), Myers Foreman (Lamar University), Olac Fuentes (University of Texas at El Paso), Edward F. Gehringer (North Carolina State University), Harold Grossman (Clemson University), Barbara Guillot (Louisiana State University), Stuart hansen (University of Wisconsin, Parkside), Dan Harvey (Southern Oregon University), Ron Hofman (Red River College, Canada), Stephen Hughes (Roanoke College), Vladan Jovanovic (Georgia Southern University), Edwin Kay (Lehigh University), Larry King (University of Texas at Dallas), Nana Kofi (Langara College, Canada), George Koutsogiannakis (Illinois Institute of Technology), Roger Kraft (Purdue University at Calumet), Norman Krumpe (Miami University), Hong Lin (DeVry Institute), Dan Lipsa (Armstrong Atlantic State University), James Madison (Rensselaer Polytechnic Institute), Frank Malinowski (Darton College), Tim Margush (University of Akron), Debbie Masada (Sun Microsystems), Blayne Mayfield (Oklahoma State University), John McGrath (J.P. McGrath Consulting), Hugh McGuire (Grand Valley State), Shyamal Mitra (University of Texas at Austin), Michel Mitri (James Madison University), Kenrick

Mock (University of Alaska Anchorage), Frank Murgolo (California State University, Long Beach), Jun Ni (University of Iowa), Benjamin Nystuen (University of Colorado at Colorado Springs), Maureen Opkins (CA State University, Long Beach), Gavin Osborne (University of Saskatchewan), Kevin Parker (Idaho State University), Dale Parson (Kutztown University), Mark Pendergast (Florida Gulf Coast University), Richard Povinelli (Marquette University), Roger Priebe (University of Texas at Austin), Mary Ann Pumphrey (De Anza Junior College), Pat Roth (Southern Polytechnic State University), Amr Sabry (Indiana University), Ben Setzer (Kennesaw State University), Carolyn Schauble (Colorado State University), David Scuse (University of Manitoba), Ashraf Shirani (San Jose State University), Daniel Spiegel (Kutztown University), Joslyn A. Smith (Florida Atlantic University), Lixin Tao (Pace University), Ronald F. Taylor (Wright State University), Russ Tront (Simon Fraser University), Deborah Trytten (University of Oklahoma), Michael Verdicchio (Citadel), Kent Vidrine (George Washington University), Bahram Zartoshty (California State University at Northridge).

能够与 Pearson 出版社一起工作,我感到非常愉快和荣幸。感谢 Tracy Johnson 和她的同事 Marcia Horton、Yez Alayan、Carole Snyder、Scott Disanno、Bob Engelhardt、Haseen Khan, 感谢他们组织、开展和积极促进本项目。

一如既往,感谢我妻子 Samantha 的爱、支持和鼓励。

出版者的话
中文版序
译者序
前言

第 19 章 泛型 1

19.1 引言 1

19.2 动机和优点 1

19.3 定义泛型类和接口 4

19.4 泛型方法 5

19.5 示例学习：对一个对象数组
进行排序 7

19.6 原始类型和向后兼容 8

19.7 通配泛型 10

19.8 消除泛型和对泛型的限制 12

19.9 示例学习：泛型矩阵类 15

关键术语 19

本章小结 19

测试题 20

编程练习题 20

**第 20 章 线性表、栈、队列和
优先队列** 21

20.1 引言 21

20.2 合集 21

20.3 迭代器 25

20.4 线性表 26

20.4.1 List 接口中的通用方法 26

20.4.2 数组线性表类 ArrayList 和
链表类 LinkedList 27

20.5 Comparator 接口 30

20.6 线性表和合集的静态方法 32

20.7 示例学习：弹球 35

20.8 向量类和栈类 38

20.9 队列和优先队列 40

20.9.1 Queue 接口 40

20.9.2 双端队列 Deque 和链表
LinkedList 40

20.10 示例学习：表达式求值 43

关键术语 47

本章小结 47

测试题 47

编程练习题 47

第 21 章 集合和映射表 53

21.1 引言 53

21.2 集合 53

21.2.1 HashSet 54

21.2.2 LinkedHashSet 57

21.2.3 TreeSet 58

21.3 比较集合和线性表的性能 61

21.4 示例学习：统计关键字 63

21.5 映射表 65

21.6 示例学习：单词的出现次数 69

21.7 单元素与不可变的合集
和映射表 71

关键术语 72

本章小结 72

测试题 72

编程练习题 72

第 22 章 开发高效算法 75

22.1 引言 75

22.2 使用大 O 符号来衡量算法
效率 75

22.3 示例：确定大 O 77

22.4 分析算法的时间复杂度	81	23.8.2 实现阶段 II	132
22.4.1 分析二分查找算法	81	23.8.3 结合两个阶段	133
22.4.2 分析选择排序算法	81	23.8.4 外部排序复杂度	136
22.4.3 分析汉诺塔问题	81	关键术语	136
22.4.4 常用的递推关系	82	本章小结	136
22.4.5 比较常用的增长函数	82	测试题	137
22.5 使用动态编程计算 斐波那契数	83	编程练习题	137
22.6 使用欧几里得算法求、 最大公约数	85	第 24 章 实现线性表、栈、队列和 优先队列	141
22.7 寻找素数的高效算法	89	24.1 引言	141
22.8 使用分而治之法寻找 最近的点对	94	24.2 线性表的通用特性	141
22.9 使用回溯法解决八皇后问题	97	24.3 数组线性表	144
22.10 计算几何：寻找凸包	99	24.4 链表	151
22.10.1 卷包裹算法	100	24.4.1 结点	151
22.10.2 格雷厄姆算法	101	24.4.2 MyLinkedList 类	153
关键术语	102	24.4.3 实现 MyLinkedList	154
本章小结	102	24.4.4 MyArrayList 和 MyLinkedList	162
测试题	103	24.4.5 链表的变体	162
编程练习题	103	24.5 栈和队列	163
第 23 章 排序	109	24.6 优先队列	167
23.1 引言	109	本章小结	168
23.2 插入排序	110	测试题	169
23.3 冒泡排序	112	编程练习题	169
23.4 归并排序	114	第 25 章 二叉查找树	171
23.5 快速排序	117	25.1 引言	171
23.6 堆排序	121	25.2 二叉查找树	171
23.6.1 堆的存储	122	25.2.1 表示二叉查找树	172
23.6.2 添加一个新的结点	122	25.2.2 查找一个元素	173
23.6.3 删除根结点	123	25.2.3 在 BST 中插入一个元素	173
23.6.4 Heap 类	124	25.2.4 树的遍历	174
23.6.5 使用 Heap 类进行排序	126	25.2.5 BST 类	176
23.6.6 堆排序的时间复杂度	127	25.3 删除 BST 中的一个元素	184
23.7 桶排序和基数排序	128	25.4 树的可视化和 MVC	189
23.8 外部排序	129	25.5 迭代器	192
23.8.1 实现阶段 I	131	25.6 示例学习：数据压缩	194

关键术语	199	编程练习题	242
本章小结	199		
测试题	199	第 28 章 图及其应用	244
编程练习题	199	28.1 引言	244
第 26 章 AVL 树	203	28.2 基本的图术语	245
26.1 引言	203	28.3 表示图	247
26.2 重新平衡树	204	28.3.1 表示顶点	247
26.3 为 AVL 树设计类	205	28.3.2 表示边: 边数组	248
26.4 重写 insert 方法	207	28.3.3 表示边: Edge 对象	248
26.5 实现旋转	207	28.3.4 表示边: 邻接矩阵	249
26.6 实现 delete 方法	208	28.3.5 表示边: 邻接线性表	249
26.7 AVLTree 类	209	28.4 图建模	251
26.8 测试 AVLTree 类	214	28.5 图的可视化	261
26.9 AVL 树的时间复杂度分析	216	28.6 图的遍历	263
关键术语	217	28.7 深度优先搜索 (DFS)	264
本章小结	217	28.7.1 DFS 的算法	264
测试题	217	28.7.2 DFS 的实现	265
编程练习题	217	28.7.3 DFS 的应用	267
第 27 章 散列	219	28.8 示例学习: 连通圆问题	268
27.1 引言	219	28.9 广度优先搜索 (BFS)	270
27.2 什么是散列	219	28.9.1 BFS 的算法	270
27.3 散列函数和散列码	220	28.9.2 BFS 的实现	271
27.3.1 基本数据类型的散列码	220	28.9.3 BFS 的应用	272
27.3.2 字符串类型的散列码	221	28.10 示例学习: 9 枚硬币	
27.3.3 压缩散列码	221	反面问题	273
27.4 使用开放地址法处理冲突	222	关键术语	278
27.4.1 线性探测	222	本章小结	278
27.4.2 二次探测法	223	测试题	278
27.4.3 再哈希法	224	编程练习题	278
27.5 使用链地址法处理冲突	225		
27.6 装填因子和再散列	226	第 29 章 加权图及其应用	283
27.7 使用散列实现映射表	227	29.1 引言	283
27.8 使用散列实现集合	235	29.2 加权图的表示	284
关键术语	241	29.2.1 加权边的表示: 边数组	284
本章小结	242	29.2.2 加权邻接矩阵	285
测试题	242	29.2.3 邻接线性表	285
		29.3 WeightedGraph 类	286
		29.4 最小生成树	292

29.4.1 最小生成树算法	293	31.2.1 服务器套接字	351
29.4.2 完善 Prim 的 MST 算法	295	31.2.2 客户端套接字	351
29.4.3 MST 算法的实现	295	31.2.3 通过套接字进行 数据传输	352
29.5 寻找最短路径	298	31.2.4 客户端 / 服务器示例	353
29.6 示例学习: 加权的 9 枚硬币 反面问题	305	31.3 InetAddress 类	357
关键术语	308	31.4 服务多个客户	358
本章小结	308	31.5 发送和接收对象	361
测试题	309	31.6 示例学习: 分布式井字游戏	365
编程练习题	309	关键术语	376
第 30 章 多线程和并行程序设计	314	本章小结	376
30.1 引言	314	测试题	376
30.2 线程的概念	314	编程练习题	376
30.3 创建任务和线程	315	第 32 章 Java 数据库程序设计	379
30.4 Thread 类	318	32.1 引言	379
30.5 示例学习: 闪烁的文本	320	32.2 关系型数据库系统	379
30.6 线程池	322	32.2.1 关系结构	380
30.7 线程同步	324	32.2.2 完整性约束	381
30.7.1 synchronized 关键字	326	32.3 SQL	383
30.7.2 同步语句	327	32.3.1 在 MySQL 上创建 用户账户	383
30.8 利用加锁同步	327	32.3.2 创建数据库	384
30.9 线程间协作	329	32.3.3 创建和删除表	385
30.10 示例学习: 生产者 / 消费者	333	32.3.4 简单插入、更新和删除	386
30.11 阻塞队列	336	32.3.5 简单查询	387
30.12 信号量	338	32.3.6 比较运算符和 布尔运算符	387
30.13 避免死锁	339	32.3.7 操作符 like、between-and 和 is null	388
30.14 线程状态	340	32.3.8 列的别名	388
30.15 同步合集	341	32.3.9 算术运算符	389
30.16 并行编程	342	32.3.10 显示互不相同的记录	389
关键术语	346	32.3.11 显示排好序的记录	390
本章小结	346	32.3.12 联结表	390
测试题	347	32.4 JDBC	391
编程练习题	347	32.4.1 使用 JDBC 开发数据库 应用程序	392
第 31 章 网络	350		
31.1 引言	350		
31.2 客户端 / 服务器计算	351		

32.4.2 从 JavaFX 访问数据库	396	33.5 示例学习：计算器	425
32.5 PreparedStatement	398	33.6 会话跟踪	428
32.6 CallableStatement	400	33.7 验证输入	430
32.7 获取元数据	403	33.8 将数据库与 facelet 绑定	434
32.7.1 数据库元数据	403	33.9 打开一个新的 JSF 页面	439
32.7.2 获取数据库表	404	关键术语	445
32.7.3 结果集元数据	405	本章小结	445
关键术语	406	测试题	445
本章小结	406	编程练习题	446
测试题	407		
编程练习题	407	附录A Java关键字	451
第 33 章 JavaServer Faces	411	附录B ASCII字符集	452
33.1 引言	411	附录C 操作符优先级表	453
33.2 开始使用 JSF	411	附录D Java修饰符	454
33.2.1 创建一个 JSF 项目	412	附录E 特殊浮点值	455
33.2.2 一个基本的 JSF 页面	412	附录F 数系	456
33.2.3 JSF 的受管 JavaBean	414	附录G 位操作	460
33.2.4 JSF 表达式	416	附录H 正则表达式	461
33.3 JSF GUI 组件	418	附录I 枚举类型	465
33.4 处理表单	421		

泛 型

【】 教学目标

- 描述泛型的优点 (19.2 节)。
- 使用泛型类和接口 (19.2 节)。
- 定义泛型类和接口 (19.3 节)。
- 解释为什么泛型类型可以提高可靠性和可读性 (19.3 节)。
- 定义并使用泛型方法和受限泛型类型 (19.4 节)。
- 开发一个泛型排序方法来对任意一个 `Comparable` 对象数组排序 (19.5 节)。
- 使用原始类型以向后兼容 (19.6 节)。
- 解释为什么需要通配泛型 (19.7 节)。
- 描述泛型类型消除并列出一些由类型消除引起的泛型类型的限制和局限性 (19.8 节)。
- 设计并实现泛型矩阵类 (19.9 节)。

19.1 引言

🔑 要点提示：泛型可以使我们在编译时而不是在运行时检测出错误。

你已经在第 11 章使用了一个泛型类 `ArrayList`，在第 13 章使用了一个泛型接口 `Comparable`。泛型 (generic) 可以参数化类型。这个能力使我们可以定义带泛型类型的类或方法，随后编译器会用具体的类型来替换它。例如，Java 定义了一个泛型类 `ArrayList` 用于存储泛型类型的元素。基于这个泛型类，可以创建用于保存字符串的 `ArrayList` 对象，以及保存数字的 `ArrayList` 对象。这里，字符串和数字是取代泛型类型的具体类型。

使用泛型的主要优点是能够在编译时而不是在运行时检测出错误。泛型类或方法允许用户指定可以和这些类或方法一起工作的对象类型。如果试图使用一个不相容的对象，编译器就会检测出这个错误。

本章介绍如何定义和使用泛型类、泛型接口和泛型方法，并且展示如何使用泛型来提高软件的可靠性和可读性。本章可以和第 13 章一起学习。

19.2 动机和优点

🔑 要点提示：使用 Java 泛型的动机是在编译时检测出错误。

从 JDK 1.5 开始，Java 允许定义泛型类、泛型接口和泛型方法。Java API 中的一些接口和类使用泛型进行了修改。例如，在 JDK 1.5 之前，`java.lang.Comparable` 接口被定义为如图 19-1a 所示，但是，在 JDK 1.5 以后它被修改为如图 19-1b 所示。

这里的 `<T>` 表示形式泛型类型 (formal generic type)，随后可以用一个实际具体类型 (actual concrete type) 来替换它。替换泛型类型称为泛型实例化 (generic instantiation)。按照惯例，像 `E` 或 `T` 这样的单个大写字母用于表示形式泛型类型。

```
package java.lang;

public interface Comparable {
    public int compareTo(Object o)
}
```

a) JDK 1.5 之前

```
package java.lang;

public interface Comparable<T> {
    public int compareTo(T o)
}
```

b) JDK 1.5

图 19-1 从 JDK 1.5 开始, 使用泛型类型重新定义 java.lang.Comparable 接口

为了了解使用泛型的好处, 我们来检查图 19-2 中的代码。图 19-2a 中的语句将 `c` 声明为一个引用变量, 它的类型是 `Comparable`, 然后调用 `compareTo` 方法来比较 `Date` 对象和一个字符串。这样的代码可以编译, 但是它会产生一个运行时错误, 因为字符串不能与 `Date` 对象进行比较。

```
Comparable c = new Date();
System.out.println(c.compareTo("red"));
```

a) JDK 1.5 之前

```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

b) JDK 1.5

图 19-2 新泛型类型在编译时检测到可能的错误

图 19-2b 中的语句将 `c` 声明为一个引用变量, 它的类型是 `Comparable<Date>`, 然后调用 `compareTo` 方法来比较 `Date` 对象和一个字符串。这样的代码会产生编译错误, 因为传递给 `compareTo` 方法的参数必须是 `Date` 类型的。由于这个错误可以在编译时而不是运行时被检测到, 因而泛型类型使程序更加可靠。

在 11.11 节中介绍过 `ArrayList` 类, 从 JDK 1.5 开始, 该类是一个泛型类。图 19-3 分别给出 `ArrayList` 类在 JDK 1.5 之前和从 JDK 1.5 开始的类图。

```
java.util.ArrayList

+ArrayList()
+add(o: Object): void
+add(index: int, o: Object): void
+clear(): void
+contains(o: Object): boolean
+get(index: int): Object
+indexOf(o: Object): int
+isEmpty(): boolean
+lastIndexOf(o: Object): int
+remove(o: Object): boolean
+size(): int
+remove(index: int): boolean
+set(index: int, o: Object): Object
```

a) JDK 1.5 之前的 ArrayList

```
java.util.ArrayList<E>

+ArrayList()
+add(o: E): void
+add(index: int, o: E): void
+clear(): void
+contains(o: Object): boolean
+get(index: int): E
+indexOf(o: Object): int
+isEmpty(): boolean
+lastIndexOf(o: Object): int
+remove(o: Object): boolean
+size(): int
+remove(index: int): boolean
+set(index: int, o: E): E
```

b) 从 JDK 1.5 开始的 ArrayList

图 19-3 从 JDK 1.5 开始, ArrayList 是一个泛型类

例如, 下面的语句创建一个字符串的线性表:

```
ArrayList<String> list = new ArrayList<>();
```

现在, 就只能向该线性表中添加字符串。例如,

```
list.add("Red");
```

如果试图向其中添加非字符串，就会产生编译错误。例如，下面的语句就是不合法的，因为 `list` 只能包含字符串：

```
list.add(new Integer(1));
```

泛型类型必须是引用类型。不能使用 `int`、`double` 或 `char` 这样的基本类型来替换泛型类型。例如，下面的语句是错误的：

```
ArrayList<int> intList = new ArrayList<>();
```

为了给 `int` 值创建一个 `ArrayList` 对象，必须使用

```
ArrayList<Integer> intList = new ArrayList<>();
```

可以在 `intList` 中加入一个 `int` 值。例如，

```
intList.add(5);
```

Java 会自动地将 5 包装为 `new Integer(5)`。这个过程称为自动打包（autoboxing），这是在 10.8 节中介绍的。

无须类型转换就可以从一个元素类型已指定的线性表中获取一个值，因为编译器已经知道了这个元素类型。例如，下面的语句创建了一个包含字符串的线性表，然后将字符串加入这个线性表，最后从这个线性表中获取该字符串。

```
1 ArrayList<String> list = new ArrayList<>();
2 list.add("Red");
3 list.add("White");
4 String s = list.get(0); // No casting is needed
```

在 JDK 1.5 之前，由于没有使用泛型，所以必须把返回值的类型转换为 `String`，如下所示：

```
String s = (String)(list.get(0)); // Casting needed prior to JDK 1.5
```

如果元素是包装类型，例如，`Integer`、`Double` 或 `Character`，那么可以直接将这个元素赋给一个基本类型的变量。这个过程称为自动拆箱（autounboxing），这是在 10.8 节中介绍的。例如，请看下面的代码：

```
1 ArrayList<Double> list = new ArrayList<>();
2 list.add(5.5); // 5.5 is automatically converted to new Double(5.5)
3 list.add(3.0); // 3.0 is automatically converted to new Double(3.0)
4 Double doubleObject = list.get(0); // No casting is needed
5 double d = list.get(1); // Automatically converted to double
```

在第 2 行和第 3 行，5.5 和 3.0 自动转换为 `Double` 对象，并添加到 `list` 中。在第 4 行，`list` 中的第一个元素被赋给一个 `Double` 变量。在此无须类型转换，因为 `list` 被声明为 `Double` 对象。在第 5 行，`list` 中的第二个元素被赋给一个 `double` 变量。`list.get(1)` 中的对象自动转换为一个基本类型的值。

✓ 复习题

19.1 图 a 和图 b 中有编译错误吗？

```
ArrayList dates = new ArrayList();
dates.add(new Date());
dates.add(new String());
```

a) JDK 1.5 之前

```
ArrayList<Date> dates =
    new ArrayList<>();
dates.add(new Date());
dates.add(new String());
```

b) 从 JDK 1.5 开始

19.2 图 a 中有什么错误？图 b 中的代码正确吗？

```
ArrayList dates = new ArrayList();
dates.add(new Date());
Date date = dates.get(0);
```

a) JDK 1.5 之前

```
ArrayList<Date> dates =
    new ArrayList<>();
dates.add(new Date());
Date date = dates.get(0);
```

b) 从 JDK 1.5 开始

19.3 使用泛型类型的优势是什么？

19.3 定义泛型类和接口

🔑 **要点提示：**可以为类或者接口定义泛型。当使用类来创建对象，或者使用类或接口来声明引用变量时，必须指定具体的类型。

我们修改 11.13 节中的栈类，将元素类型通用化为泛型。新的名为 `GenericStack` 的栈类如图 19-4 所示，在程序清单 19-1 中实现它。

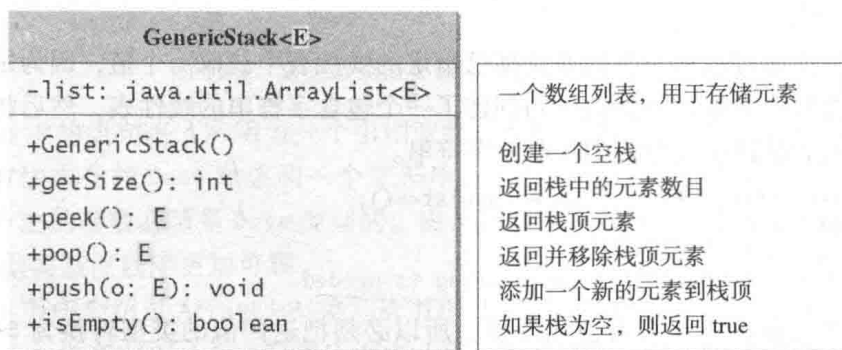


图 19-4 `GenericStack` 类封装了栈的存储，并提供使用该栈的操作

程序清单 19-1 `GenericStack.java`

```
1 public class GenericStack<E> {
2     private java.util.ArrayList<E> list = new java.util.ArrayList<>();
3
4     public int getSize() {
5         return list.size();
6     }
7
8     public E peek() {
9         return list.get(getSize() - 1);
10    }
11
12    public void push(E o) {
13        list.add(o);
14    }
15
16    public E pop() {
17        E o = list.get(getSize() - 1);
18        list.remove(getSize() - 1);
19        return o;
20    }
21
22    public boolean isEmpty() {
23        return list.isEmpty();
24    }
25 }
```

```
25
26     @Override
27     public String toString() {
28         return "stack: " + list.toString();
29     }
30 }
```

下面的例子中，先创建一个存储字符串的栈，然后向这个栈添加三个字符串；

```
GenericStack<String> stack1 = new GenericStack<>();
stack1.push("London");
stack1.push("Paris");
stack1.push("Berlin");
```

该示例创建一个存储整数的栈，然后向这个栈添加三个整数：

```
GenericStack<Integer> stack2 = new GenericStack<>();
stack2.push(1); // autoboxing 1 to new Integer(1)
stack2.push(2);
stack2.push(3);
```

可以不使用泛型，而将元素类型设置为 `Object`，也可以容纳任何对象类型。但是，使用泛型能够提高软件的可靠性和可读性，因为某些错误能在编译时而不是运行时被检测到。例如，由于 `stack1` 被声明为 `GenericStack<String>`，所以，只可以将字符串添加到这个栈中。如果试图向 `stack1` 中添加整数就会产生编译错误。

{ } 警告：为了创建一个字符串堆栈，可以使用 `new GenericStack<String>()` 或 `new GenericStack<>()`。这可能会误导你认为 `GenericStack` 的构造方法应该定义为

```
public GenericStack<E>()
```

这是错误的。它应该被定义为

```
public GenericStack()
```

{ } 注意：有时候，泛型类可能会有多个参数。在这种情况下，应将所有参数一起放在尖括号中，并用逗号分隔开，比如 `<E1,E2,E3>`。

{ } 注意：可以定义一个类或接口作为泛型类或者泛型接口的子类型。例如，在 Java API 中，`java.lang.String` 类被定义为实现 `Comparable` 接口，如下所示：

```
public class String implements Comparable<String>
```

✓ 复习题

19.4 Java API 中，`java.lang.Comparable` 的泛型定义是什么？

19.5 既然使用 `new ArrayList<String>()` 创建了字符串的 `ArrayList` 的一个实例，那么应该将 `ArrayList` 类的构造方法定义为如下所示吗？

```
public ArrayList<E>()
```

19.6 泛型类可以拥有多个泛型参数吗？

19.7 在类中如何声明一个泛型类型？

19.4 泛型方法

🔑 要点提示：可以为静态方法定义泛型类型。

可以定义泛型接口（例如，图 19-1b 中的 `Comparable` 接口）和泛型类（例如，程序清单

19-1 中的 `GenericStack` 类), 也可以使用泛型类型来定义泛型方法。例如, 程序清单 19-2 定义了一个泛型方法 `print` (第 10 ~ 14 行) 来打印对象数组。第 6 行传递一个整数对象的数组来调用泛型方法 `print`。第 7 行用字符串数组调用 `print`。

程序清单 19-2 `GenericMethodDemo.java`

```

1 public class GenericMethodDemo {
2     public static void main(String[] args ) {
3         Integer[] integers = {1, 2, 3, 4, 5};
4         String[] strings = {"London", "Paris", "New York", "Austin"};
5
6         GenericMethodDemo.<Integer>print(integers);
7         GenericMethodDemo.<String>print(strings);
8     }
9
10    public static <E> void print(E[] list) {
11        for (int i = 0; i < list.length; i++)
12            System.out.print(list[i] + " ");
13        System.out.println();
14    }
15 }
```

为了声明泛型方法, 将泛型类型 `<E>` 置于方法头中关键字 `static` 之后。例如,

```
public static <E> void print(E[] list)
```

为了调用泛型方法, 需要将实际类型放在尖括号内作为方法名的前缀。例如,

```
GenericMethodDemo.<Integer>print(integers);
GenericMethodDemo.<String>print(strings);
```

或者如下简单调用:

```
print(integers);
print(strings);
```

在后面一种情况中, 实际类型没有明确指定。编译器自动发现实际类型。

可以将泛型指定为另外一种类型的子类型。这样的泛型类型称为受限的 (bounded)。例如, 程序清单 19-3 修改了程序清单 13-4 中的 `equalArea` 方法, 以测试两个几何对象是否具有相同的面积。受限的泛型类型 `<E extends GeometricObject>` (第 10 行) 将 `E` 指定为 `GeometricObject` 的泛型子类型。必须传递两个 `GeometricObject` 的实例来调用 `equalArea`。

程序清单 19-3 `BoundedTypeDemo.java`

```

1 public class BoundedTypeDemo {
2     public static void main(String[] args ) {
3         Rectangle rectangle = new Rectangle(2, 2);
4         Circle circle = new Circle(2);
5
6         System.out.println("Same area? " +
7             equalArea(rectangle, circle));
8     }
9
10    public static <E extends GeometricObject> boolean equalArea(
11        E object1, E object2) {
12        return object1.getArea() == object2.getArea();
13    }
14 }
```

注意: 非受限泛型类型 `<E>` 等同于 `<E extends Object>`。

注意: 为了定义一个类为泛型类型, 需要将泛型类型放在类名之后, 例如, `GenericStack<E>`。

为了定义一个方法为泛型类型，要将泛型类型放在方法返回类型之前，例如，`<E> void max (E o1,E o2)`。

复习题

19.8 如何声明一个泛型方法？如何调用一个泛型方法？

19.9 什么是受限泛型类型？

19.5 示例学习：对一个对象数组进行排序

要点提示：可以开发一个泛型方法，对一个 `Comparable` 对象数组进行排序。

本节提供一个泛型方法，对一个 `Comparable` 对象数组进行排序。这些对象是 `Comparable` 接口的实例，它们使用 `compareTo` 方法进行比较。为了测试该方法，程序对一个整数数组、一个双精度数字数组、一个字符数组以及一个字符串数组分别进行了排序。程序如程序清单 19-4 所示。

程序清单 19-4 GenericSort.java

```
1 public class GenericSort {
2     public static void main(String[] args) {
3         // Create an Integer array
4         Integer[] intArray = {new Integer(2), new Integer(4),
5                               new Integer(3)};
6
7         // Create a Double array
8         Double[] doubleArray = {new Double(3.4), new Double(1.3),
9                                  new Double(-22.1)};
9
10
11        // Create a Character array
12        Character[] charArray = {new Character('a'),
13                                  new Character('J'), new Character('r')};
14
15        // Create a String array
16        String[] stringArray = {"Tom", "Susan", "Kim"};
17
18        // Sort the arrays
19        sort(intArray);
20        sort(doubleArray);
21        sort(charArray);
22        sort(stringArray);
23
24        // Display the sorted arrays
25        System.out.print("Sorted Integer objects: ");
26        printList(intArray);
27        System.out.print("Sorted Double objects: ");
28        printList(doubleArray);
29        System.out.print("Sorted Character objects: ");
30        printList(charArray);
31        System.out.print("Sorted String objects: ");
32        printList(stringArray);
33    }
34
35    /** Sort an array of comparable objects */
36    public static <E extends Comparable<E>> void sort(E[] list) {
37        E currentMin;
38        int currentMinIndex;
39
40        for (int i = 0; i < list.length - 1; i++) {
41            // Find the minimum in the list[i+1..list.length-2]
42            currentMin = list[i];
43            currentMinIndex = i;
```

```

44
45     for (int j = i + 1; j < list.length; j++) {
46         if (currentMin.compareTo(list[j]) > 0) {
47             currentMin = list[j];
48             currentMinIndex = j;
49         }
50     }
51
52     // Swap list[i] with list[currentMinIndex] if necessary;
53     if (currentMinIndex != i) {
54         list[currentMinIndex] = list[i];
55         list[i] = currentMin;
56     }
57 }
58 }
59
60 /** Print an array of objects */
61 public static void printList(Object[] list) {
62     for (int i = 0; i < list.length; i++)
63         System.out.print(list[i] + " ");
64     System.out.println();
65 }
66 }

```

```

Sorted Integer objects: 2 3 4
Sorted Double objects: -22.1 1.3 3.4
Sorted Character objects: J a r
Sorted String objects: Kim Susan Tom

```


sort 方法的算法和程序清单 7-8 中的一样。那个程序中的 sort 方法对一个 double 数值的数组进行了排序。本例中的 sort 方法可以对任何对象类型的数组进行排序，只要这些对象也是 Comparable 接口的实例。泛型类型定义为 `<E extends Comparable <E>>` (第 36 行)。这具有两个含义：首先，它指定 E 是 Comparable 的子类型；其次，它还指定进行比较的元素是 E 类型的。

sort 方法使用 compareTo 方法来确定数组中对象的排序 (第 46 行)。Integer、Double、Character 以及 String 实现 Comparable。因此这些类的对象可以使用 compareTo 方法进行比较。程序创建一个 Integer 对象数组、一个 Double 对象数组、一个 Character 对象数组以及一个 String 对象数组 (第 4 ~ 16 行)，然后调用 sort 方法来对这些数组进行排序 (第 19 ~ 22 行)。

✓ 复习题

- 19.10 给出 `int[] list = {1, 2, -1}`，可以使用程序清单 19-4 中的 sort 方法调用 `sort(list)` 吗？
- 19.11 给出 `int[] list = {new Integer(1), new Integer(2), new Integer(-1)}`，可以使用程序清单 19-4 中的 sort 方法调用 `sort(list)` 吗？

19.6 原始类型和向后兼容

 **要点提示：**没有指定具体类型的泛型类和泛型接口被称为原始类型，用于和早期的 Java 版本向后兼容。

可以使用泛型类而无须指定具体类型，如下所示：

```
GenericStack stack = new GenericStack(); // raw type
```

它大体等价于下面的语句：


```
GenericStack<Object> stack = new GenericStack<Object>();
```

像这样不带类型参数的 `GenericStack` 和 `ArrayList` 泛型类称为原始类型 (raw type)。使用原始类型可以向后兼容 Java 的早期版本。例如, 从 JDK 1.5 开始, 在 `java.lang.Comparable` 中使用了泛型类型, 但是, 许多代码仍然使用原始类型 `Comparable`, 如程序清单 19-5 所示。

程序清单 19-5 Max.java

```
1 public class Max {
2     /** Return the maximum of two objects */
3     public static Comparable max(Comparable o1, Comparable o2) {
4         if (o1.compareTo(o2) > 0)
5             return o1;
6         else
7             return o2;
8     }
9 }
```

`Comparable o1` 和 `Comparable o2` 都是原始类型声明。但是小心: 原始类型是不安全的。例如, 你可能会使用下面的语句调用 `max` 方法:

```
Max.max("Welcome", 23); // 23 is autoboxed into new Integer(23)
```

这会引起一个运行时错误, 因为不能将字符串与整数对象进行比较。如果在编译时使用了选项 `-Xlint:unchecked`, Java 编译器就会对第 3 行显示一条警告, 如图 19-5 所示。

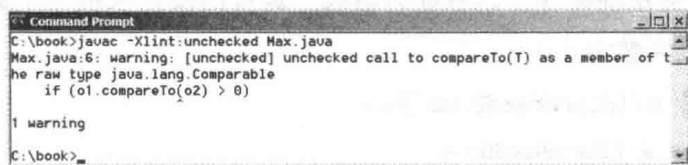


图 19-5 使用编译器选项 `-Xlint:unchecked` 会显示一条免检的警告

编写 `max` 方法的更好方式是使用泛型类型, 如程序清单 19-6 所示。

程序清单 19-6 MaxUsingGenericType.java

```
1 public class MaxUsingGenericType {
2     /** Return the maximum of two objects */
3     public static <E extends Comparable<E>> E max(E o1, E o2) {
4         if (o1.compareTo(o2) > 0)
5             return o1;
6         else
7             return o2;
8     }
9 }
```

如果使用下面的命令调用 `max` 方法:

```
// 23 is autoboxed into new Integer(23)
MaxUsingGenericType.max("Welcome", 23);
```

就会显示一个编译错误, 因为 `MaxUsingGenericType` 中的 `max` 方法的两个参数必须是相同的类型 (例如, 两个字符串或两个整数对象)。此外, 类型 `E` 必须是 `Comparable<E>` 的子类型。

下面的代码是另外一个例子, 可以在第 1 行声明一个原始类型 `stack`, 在第 2 行将 `new GenericStack<String>` 赋给它, 然后在第 3 行和第 4 行将一个字符串和一个整数对象压入

栈中。

```
1 GenericStack stack;
2 stack = new GenericStack<String>();
3 stack.push("Welcome to Java");
4 stack.push(new Integer(2));
```

然而，第 4 行是不安全的，因为该栈是用于存储字符串的，但是一个 `Integer` 对象被添加到该栈中。第 3 行本应是可行的，但是编译器会在第 3 行和第 4 行都显示警告，因为它不能理解程序的语义。编译器所知道的就是该栈是一个原始类型，并且在执行某些操作时会不安全。因此，它会显示警告以提醒潜在的问题。

提示：由于原始类型是不安全的，所以，本书从此不再使用原始类型。

复习题

19.12 什么是原始类型？为什么原始类型是不安全的？为什么 Java 中允许使用原始类型？

19.13 使用什么样的语法来声明一个使用原始类型的 `ArrayList` 引用变量，以及将一个原始类型的 `ArrayList` 对象赋值给该变量？

19.7 通配泛型

要点提示：可以使用非受限通配、受限通配或者下限通配来对一个泛型类型指定范围。

通配泛型是什么？为什么需要通配泛型？程序清单 19-7 给出了一个例子，以展示为什么需要通配泛型。该例子定义了一个泛型 `max` 方法，该方法可以找出数字栈中的最大数（第 12 ~ 22 行）。`main` 方法创建了一个整数对象栈，然后向该栈添加三个整数，最后调用 `max` 方法找出该栈中的最大数字。

程序清单 19-7 WildCardNeedDemo.java

```
1 public class WildCardNeedDemo {
2     public static void main(String[] args) {
3         GenericStack<Integer> intStack = new GenericStack<>();
4         intStack.push(1); // 1 is autoboxed into new Integer(1)
5         intStack.push(2);
6         intStack.push(-2);
7
8         System.out.print("The max number is " + max(intStack));
9     }
10
11     /** Find the maximum in a stack of numbers */
12     public static double max(GenericStack<Number> stack) {
13         double max = stack.pop().doubleValue(); // Initialize max
14
15         while (!stack.isEmpty()) {
16             double value = stack.pop().doubleValue();
17             if (value > max)
18                 max = value;
19         }
20
21         return max;
22     }
23 }
```

程序清单 19-7 中的程序在第 8 行会出现编译错误，因为 `intStack` 不是 `GenericStack<Number>` 的实例。因此，不能调用 `max(intStack)`。

尽管 `Integer` 是 `Number` 的子类型，但是，`GenericStack<Integer>` 并不是 `GenericStack`

<Number> 的子类型。为了避免这个问题，可以使用通配泛型类型。通配泛型类型有三种形式——?、? extends T 或者 ? super T，其中 T 是泛型类型。

第一种形式 ? 称为非受限通配 (unbounded wildcard)，它和 ? extends Object 是一样的。第二种形式 ? extends T 称为受限通配 (bounded wildcard)，表示 T 或 T 的一个子类型。第三种形式 ? super T 称为下限通配 (lower-bound wildcard)，表示 T 或 T 的一个父类型。

使用下面的语句替换程序清单 19-7 中的第 12 行，就可以修复上面的错误：

```
public static double max(GenericStack<? extends Number> stack) {
```

<? extends Number> 是一个表示 Number 或 Number 的子类型的通配类型。因此，调用 max(new GenericStack<Integer>()) 或 max(new GenericStack<Double>()) 都是合法的。

程序清单 19-8 给出一个例子，它在 print 方法中使用 ? 通配符，打印栈中的对象以及清空栈。<?> 是一个通配符，表示任何一种对象类型。它等价于 <? extends Object>。如果用 GenericStack<Object> 替换 GenericStack<?>，会发生什么情况呢？这样调用 print(intStack) 会出错，因为 intStack 不是 GenericStack<Object> 的实例。请注意，尽管 Integer 是 Object 的一个子类型，但是 GenericStack<Integer> 并不是 GenericStack<Object> 的子类型。

程序清单 19-8 AnyWildcardDemo.java

```
1 public class AnyWildcardDemo {
2     public static void main(String[] args) {
3         GenericStack<Integer> intStack = new GenericStack<>();
4         intStack.push(1); // 1 is autoboxed into new Integer(1)
5         intStack.push(2);
6         intStack.push(-2);
7
8         print(intStack);
9     }
10
11     /** Prints objects and empties the stack */
12     public static void print(GenericStack<?> stack) {
13         while (!stack.isEmpty()) {
14             System.out.print(stack.pop() + " ");
15         }
16     }
17 }
```

什么时候需要 <? super T> 通配符呢？请看程序清单 19-9 中的例子。该例创建了一个字符串栈 stack1 (第 3 行) 和一个对象栈 stack2 (第 4 行)，然后调用 add(stack1, stack2) (第 8 行) 将 stack1 中的字符串添加到 stack2 中。在第 13 行使用 GenericStack<? super T> 来声明栈 stack2。如果用 <T> 代替 <? super T>，那么在第 8 行的 add(stack1, stack2) 上就会产生一个编译错误，因为 stack1 的类型为 GenericStack<String>，而 stack2 的类型为 GenericStack<Object>。<? super T> 表示类型 T 或 T 的父类型。Object 是 String 的父类型。

程序清单 19-9 SuperWildcardDemo.java

```
1 public class SuperWildcardDemo {
2     public static void main(String[] args) {
3         GenericStack<String> stack1 = new GenericStack<>();
4         GenericStack<Object> stack2 = new GenericStack<>();
5         stack2.push("Java");
6         stack2.push(2);
7         stack1.push("Sun");
```

```

8      add(stack1, stack2);
9      AnyWildCardDemo.print(stack2);
10   }
11
12   public static <T> void add(GenericStack<T> stack1,
13       GenericStack<? super T> stack2) {
14       while (!stack1.isEmpty())
15           stack2.push(stack1.pop());
16   }
17 }

```

如果第 12 ~ 13 行的方法头如下修改，程序也可以运行：

```

public static <T> void add(GenericStack<? extends T> stack1,
    GenericStack<T> stack2)

```

泛型类型和通配类型之间的继承关系在图 19-6 中进行了总结。在该图中，A 和 B 表示类或者接口，而 E 是泛型类型参数。

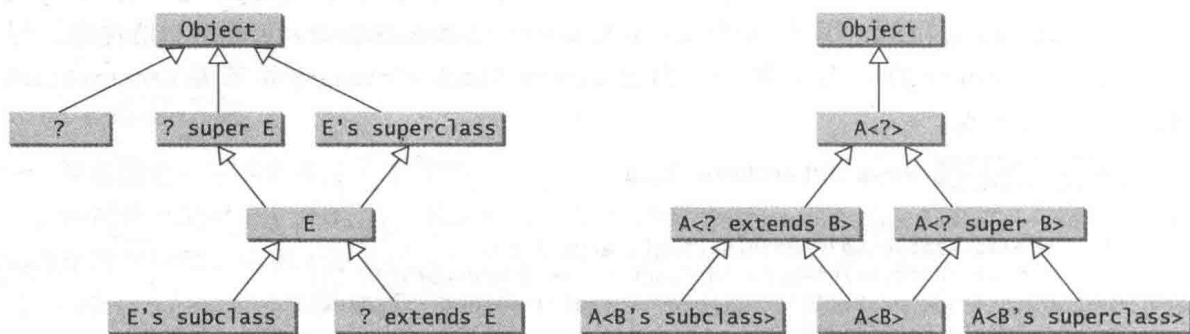


图 19-6 泛型类型和通配类型之间的关系

✓ 复习题

19.14 GenericStack 等同于 GenericStack<Object> 吗？

19.15 什么是非受限通配、受限通配、下限通配？

19.16 如果将程序清单 19-9 中的第 12 ~ 13 行改为如下所示，会发生什么情况？

```

public static <T> void add(GenericStack<T> stack1,
    GenericStack<T> stack2)

```

19.17 如果将程序清单 19-9 中的第 12 ~ 13 行改为如下所示，会发生什么情况？

```

public static <T> void add(GenericStack<? extends T> stack1,
    GenericStack<T> stack2)

```

19.8 消除泛型和对泛型的限制

要点提示：编译器可使用泛型信息，但这些信息在运行时是不可用的。这被称为类型消除。

泛型是使用一种称为类型消除（type erasure）的方法来实现的。编译器使用泛型类型信息来编译代码，但是随后会消除它。因此，泛型信息在运行时是不可用的。这种方法可以使泛型代码向后兼容使用原始类型的遗留代码。

泛型存在于编译时。一旦编译器确认泛型类型是安全使用的，就会将它转换为原始类型。例如，编译器会检查图 a 的代码里泛型是否被正确使用，然后将它翻译成如图 b 所示的在运行时使用的等价代码。图 b 中的代码使用的是原始类型。

```
ArrayList<String> list = new ArrayList<>();
list.add("Oklahoma");
String state = list.get(0);
```

a)

```
ArrayList list = new ArrayList();
list.add("Oklahoma");
String state = (String)(list.get(0));
```

b)

当编译泛型类、接口和方法时，编译器用 `Object` 类型代替泛型类型。例如，编译器会将图 a 中的方法转换为图 b 中的方法。

```
public static <E> void print(E[] list) {
    for (int i = 0; i < list.length; i++)
        System.out.print(list[i] + " ");
    System.out.println();
}
```

a)

```
public static void print(Object[] list) {
    for (int i = 0; i < list.length; i++)
        System.out.print(list[i] + " ");
    System.out.println();
}
```

b)

如果一个泛型类型是受限的，那么编译器就会用该受限类型来替换它。例如，编译器会将图 a 中的方法转换为图 b 中的方法。

```
public static <E extends GeometricObject>
    boolean equalArea(
        E object1,
        E object2) {
    return object1.getArea() ==
        object2.getArea();
}
```

a)

```
public static
    boolean equalArea(
        GeometricObject object1,
        GeometricObject object2) {
    return object1.getArea() ==
        object2.getArea();
}
```

b)

非常需要注意的是，不管实际的具体类型是什么，泛型类是被它的所有实例所共享的。假定按如下方式创建 `list1` 和 `list2`：

```
ArrayList<String> list1 = new ArrayList<>();
ArrayList<Integer> list2 = new ArrayList<>();
```

尽管在编译时 `ArrayList<String>` 和 `ArrayList<Integer>` 是两种类型，但是，在运行时只有一个 `ArrayList` 类会被加载到 JVM 中。`list1` 和 `list2` 都是 `ArrayList` 的实例，因此，下面两条语句的执行结果都为 `true`：

```
System.out.println(list1 instanceof ArrayList);
System.out.println(list2 instanceof ArrayList);
```

然而表达式 `list1 instanceof ArrayList<String>` 是错误的。由于 `ArrayList<String>` 并没有在 JVM 中存储为单独一个类，所以，在运行时使用它是毫无意义的。

由于泛型类型在运行时被消除，因此，对于如何使用泛型类型是有一些限制的。下面是其中的一些限制。

限制 1：不能使用 `new E()`

不能使用泛型类型参数创建实例。例如，下面的语句是错误的：

```
E object = new E();
```

出错的原因是运行时执行的是 `new E()`，但是运行时泛型类型 `E` 是不可用的。

限制 2：不能使用 `new E[]`

不能使用泛型类型参数创建数组。例如，下面的语句是错误的：

```
E[] elements = new E[capacity];
```

可以通过创建一个 `Object` 类型的数组，然后将它的类型转换为 `E[]` 来规避这个限制，

如下所示：

```
E[] elements = (E[])new Object[capacity];
```

但是，类型转换到 (E[]) 会导致一个免检的编译警告。该警告会出现是因为编译器无法确保在运行时类型转换是否能成功。例如，如果 E 是 String，而 new Object[] 是 Integer 对象的数组，那么 (String[])(new Object[]) 将会导致 ClassCastException 异常。这种类型的编译警告是使用 Java 泛型的不足之处，也是无法避免的。

使用泛型类创建泛型数组也是不允许的。例如，下面的代码是错误的：

```
ArrayList<String>[] list = new ArrayList<String>[10];
```

可以使用下面的代码来规避这种限制：

```
ArrayList<String>[] list = (ArrayList<String>[])new  
    ArrayList[10];
```

然而，你依然会得到一个编译警告。

限制 3：在静态上下文中不允许类的参数是泛型类型

由于泛型类的所有实例都有相同的运行时类，所以泛型类的静态变量和方法是被它的所有实例所共享的。因此，在静态方法、数据域或者初始化语句中，为类引用泛型类型参数是非法的。例如，下面的代码是非法的：

```
public class Test<E> {  
    public static void m(E o1) { // Illegal  
    }  
  
    public static E o1; // Illegal  
  
    static {  
        E o2; // Illegal  
    }  
}
```

限制 4：异常类不能是泛型的

泛型类不能扩展 java.lang.Throwable，因此，下面的类声明是非法的：

```
public class MyException<T> extends Exception {  
}
```

为什么呢？因为如果允许这样做，就应为 MyException<T> 添加一个 catch 子句，如下所示：

```
try {  
    ...  
}  
catch (MyException<T> ex) {  
    ...  
}
```

JVM 必须检查这个从 try 子句中抛出的异常以确定它是否与 catch 子句中指定的类型匹配。但这是不可能的，因为在运行时类型信息是不可得的。

✓ 复习题

19.18 什么是消除？为什么使用消除来实现 Java 泛型？

19.19 如果你的程序使用了 ArrayList<String> 和 ArrayList<Date>，JVM 会对它们都加载吗？

19.20 可以使用 new E() 为泛型类型 E 创建一个实例吗？为什么？

19.21 使用泛型类作为参数的方法可以是静态的吗？为什么？

19.22 可以定义一个自定义的泛型异常类吗？为什么？

19.9 示例学习：泛型矩阵类

要点提示：本节给出一个示例学习，使用泛型类型来设计用于矩阵运算的类。

对于所有矩阵，除了元素类型不同以外，它们的加法和乘法操作都是类似的。因此，可以设计一个父类，不管它们的元素类型是什么，该父类描述所有类型的矩阵共享的通用操作，还可以创建若干个适用于指定矩阵类型的子类。这里的示例学习给出了两种类型 `int` 和 `Rational` 的实现。对于 `int` 类型而言，包装类 `Integer` 应该用于将一个 `int` 类型的值包装到一个对象中，从而对象被传递给方法进行操作。

该类的类图如图 19-7 所示。方法 `addMatrix` 和方法 `multiplyMatrix` 将泛型类型 `E[][]` 的两个矩阵进行相加和相乘。静态方法 `printResult` 显示矩阵、操作以及它们的结果。方法 `add`、`multiply` 和 `zero` 都是抽象的，因为它们的实现依赖于数组元素的特定类型。例如，`zero()` 方法对于 `Integer` 类型返回 0，而对于 `Rational` 类型返回 0/1。这些方法将会在指定了矩阵元素类型的子类中实现。

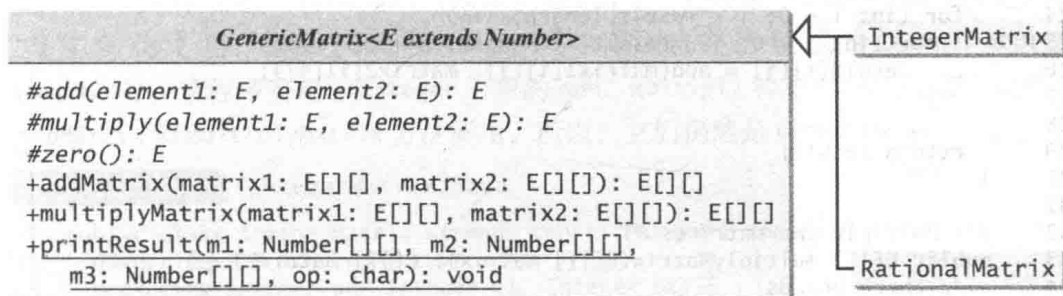


图 19-7 `GenericMatrix` 类是 `IntegerMatrix` 和 `RationalMatrix` 的一个抽象父类

`IntegerMatrix` 和 `RationalMatrix` 是 `GenericMatrix` 的具体子类。这两个类实现了在 `GenericMatrix` 类中定义的 `add`、`multiply` 和 `zero` 方法。

程序清单 19-10 实现了 `GenericMatrix` 类。第 1 行的 `<E extends Number>` 指明该泛型类型是 `Number` 的子类型。三个抽象方法 `add`、`multiply` 和 `zero` 在第 3、6 和 9 行定义。这些方法是抽象的，因为在不知道元素的确切类型时我们是不能实现它们的。`addMatrix` 方法（第 12 ~ 30 行）和 `multiplyMatrix` 方法（第 33 ~ 57 行）实现了两个矩阵的相加和相乘。所有这些方法都必须是非静态的，因为它们使用的是泛型类型 `E` 来表示类。`printResult` 方法（第 60 ~ 84 行）是静态的，因为它没有绑定到特定的实例。

矩阵元素的类型是 `Number` 的泛型子类。这样就可以使用任意 `Number` 子类的对象，只要在子类中实现了抽象方法 `add`、`multiply` 和 `zero` 即可。

`addMatrix` 和 `multiplyMatrix` 方法（第 12 ~ 57 行）是具体的方法。只要在子类中实现了 `add`、`multiply` 和 `zero` 方法，就可以使用它们。

`addMatrix` 和 `multiplyMatrix` 方法在进行操作之前检查矩阵的边界。如果两个矩阵的边界不匹配，那么程序会抛出一个异常（第 16 和 36 行）。

程序清单 19-10 GenericMatrix.java

```

1 public abstract class GenericMatrix<E extends Number> {
2     /** Abstract method for adding two elements of the matrices */
3     protected abstract E add(E o1, E o2);
4
5     /** Abstract method for multiplying two elements of the matrices */
6     protected abstract E multiply(E o1, E o2);
7
8     /** Abstract method for defining zero for the matrix element */
9     protected abstract E zero();
10
11     /** Add two matrices */
12     public E[][] addMatrix(E[][] matrix1, E[][] matrix2) {
13         // Check bounds of the two matrices
14         if ((matrix1.length != matrix2.length) ||
15             (matrix1[0].length != matrix2[0].length)) {
16             throw new RuntimeException(
17                 "The matrices do not have the same size");
18         }
19
20         E[][] result =
21             (E[][])new Number[matrix1.length][matrix1[0].length];
22
23         // Perform addition
24         for (int i = 0; i < result.length; i++)
25             for (int j = 0; j < result[i].length; j++) {
26                 result[i][j] = add(matrix1[i][j], matrix2[i][j]);
27             }
28
29         return result;
30     }
31
32     /** Multiply two matrices */
33     public E[][] multiplyMatrix(E[][] matrix1, E[][] matrix2) {
34         // Check bounds
35         if (matrix1[0].length != matrix2.length) {
36             throw new RuntimeException(
37                 "The matrices do not have compatible size");
38         }
39
40         // Create result matrix
41         E[][] result =
42             (E[][])new Number[matrix1.length][matrix2[0].length];
43
44         // Perform multiplication of two matrices
45         for (int i = 0; i < result.length; i++) {
46             for (int j = 0; j < result[0].length; j++) {
47                 result[i][j] = zero();
48
49                 for (int k = 0; k < matrix1[0].length; k++) {
50                     result[i][j] = add(result[i][j],
51                         multiply(matrix1[i][k], matrix2[k][j]));
52                 }
53             }
54         }
55
56         return result;
57     }
58
59     /** Print matrices, the operator, and their operation result */
60     public static void printResult(
61         Number[][] m1, Number[][] m2, Number[][] m3, char op) {
62         for (int i = 0; i < m1.length; i++) {

```

```

63     for (int j = 0; j < m1[0].length; j++)
64         System.out.print(" " + m1[i][j]);
65
66     if (i == m1.length / 2)
67         System.out.print(" " + op + " ");
68     else
69         System.out.print(" ");
70
71     for (int j = 0; j < m2.length; j++)
72         System.out.print(" " + m2[i][j]);
73
74     if (i == m1.length / 2)
75         System.out.print(" = ");
76     else
77         System.out.print(" ");
78
79     for (int j = 0; j < m3.length; j++)
80         System.out.print(m3[i][j] + " ");
81
82     System.out.println();
83 }
84 }
85 }

```

程序清单 19-11 实现了 `IntegerMatrix` 类。该类在第 1 行继承了 `GenericMatrix<Integer>`。在泛型实例化之后, `GenericMatrix<Integer>` 中的 `add` 方法就成为 `Integer add(Integer o1, Integer o2)`。该程序实现了 `Integer` 对象的 `add`、`multiply` 和 `zero` 方法。因为这些方法只能被 `addMatrix` 和 `multiplyMatrix` 方法调用, 所以, 它们仍然是 `protected` 的。

程序清单 19-11 `IntegerMatrix.java`

```

1  public class IntegerMatrix extends GenericMatrix<Integer> {
2      @Override /** Add two integers */
3      protected Integer add(Integer o1, Integer o2) {
4          return o1 + o2;
5      }
6
7      @Override /** Multiply two integers */
8      protected Integer multiply(Integer o1, Integer o2) {
9          return o1 * o2;
10     }
11
12     @Override /** Specify zero for an integer */
13     protected Integer zero() {
14         return 0;
15     }
16 }

```

程序清单 19-12 实现了 `RationalMatrix` 类。`Rational` 类在程序清单 13-13 中介绍过。`Rational` 是 `Number` 的子类型。`RationalMatrix` 类在第 1 行继承了 `GenericMatrix<Rational>`。在泛型实例化之后, `GenericMatrix<Rational>` 中的 `add` 方法就成为 `Rational add(Rational r1, Rational r2)`。该程序实现了 `Rational` 对象的 `add`、`multiply` 和 `zero` 方法。因为这些方法只能被 `addMatrix` 和 `multiplyMatrix` 方法调用, 所以, 它们仍然是 `protected` 的。

程序清单 19-12 `RationalMatrix.java`

```

1  public class RationalMatrix extends GenericMatrix<Rational> {
2      @Override /** Add two rational numbers */
3      protected Rational add(Rational r1, Rational r2) {
4          return r1.add(r2);
5      }

```

```

6
7  @Override /** Multiply two rational numbers */
8  protected Rational multiply(Rational r1, Rational r2) {
9      return r1.multiply(r2);
10 }
11
12 @Override /** Specify zero for a Rational number */
13 protected Rational zero() {
14     return new Rational(0, 1);
15 }
16 }

```

程序清单 19-13 给出了一个程序，该程序创建两个 `Integer` 矩阵（第 4～5 行）和一个 `IntegerMatrix` 对象（第 8 行），然后在第 12 行和第 16 行对这两个矩阵进行相加和相乘操作。

程序清单 19-13 TestIntegerMatrix.java

```

1 public class TestIntegerMatrix {
2     public static void main(String[] args) {
3         // Create Integer arrays m1, m2
4         Integer[][] m1 = new Integer[][]{{1, 2, 3}, {4, 5, 6}, {1, 1, 1}};
5         Integer[][] m2 = new Integer[][]{{1, 1, 1}, {2, 2, 2}, {0, 0, 0}};
6
7         // Create an instance of IntegerMatrix
8         IntegerMatrix integerMatrix = new IntegerMatrix();
9
10        System.out.println("\nm1 + m2 is ");
11        GenericMatrix.printResult(
12            m1, m2, integerMatrix.addMatrix(m1, m2), '+');
13
14        System.out.println("\nm1 * m2 is ");
15        GenericMatrix.printResult(
16            m1, m2, integerMatrix.multiplyMatrix(m1, m2), '*');
17    }
18 }

```

m1 + m2 is											
1	2	3		1	1	1		2	3	4	
4	5	6	+	2	2	2	=	6	7	8	
1	1	1		0	0	0		1	1	1	
m1 * m2 is											
1	2	3		1	1	1		5	5	5	
4	5	6	*	2	2	2	=	14	14	14	
1	1	1		0	0	0		3	3	3	

程序清单 19-14 给出了一个程序，该程序创建两个 `Rational` 矩阵（第 4～10 行）和一个 `RationalMatrix` 对象（第 13 行），然后在第 17 行和第 19 行对这两个矩阵进行相加和相乘操作。

程序清单 19-14 TestRationalMatrix.java

```

1 public class TestRationalMatrix {
2     public static void main(String[] args) {
3         // Create two Rational arrays m1 and m2
4         Rational[][] m1 = new Rational[3][3];
5         Rational[][] m2 = new Rational[3][3];
6         for (int i = 0; i < m1.length; i++)
7             for (int j = 0; j < m1[0].length; j++) {
8                 m1[i][j] = new Rational(i + 1, j + 5);
9                 m2[i][j] = new Rational(i + 1, j + 6);
10            }
11
12        // Create an instance of RationalMatrix

```

```

13 RationalMatrix rationalMatrix = new RationalMatrix();
14
15 System.out.println("\nm1 + m2 is ");
16 GenericMatrix.printResult(
17     m1, m2, rationalMatrix.addMatrix(m1, m2), '+');
18
19 System.out.println("\nm1 * m2 is ");
20 GenericMatrix.printResult(
21     m1, m2, rationalMatrix.multiplyMatrix(m1, m2), '*');
22 }
23 }

```

m1 + m2 is			
1/5 1/6 1/7	1/6 1/7 1/8	11/30 13/42 15/56	
2/5 1/3 2/7	+ 1/3 2/7 1/4	= 11/15 13/21 15/28	
3/5 1/2 3/7	1/2 3/7 3/8	11/10 13/14 45/56	
m1 * m2 is			
1/5 1/6 1/7	1/6 1/7 1/8	101/630 101/735 101/840	
2/5 1/3 2/7	* 1/3 2/7 1/4	= 101/315 202/735 101/420	
3/5 1/2 3/7	1/2 3/7 3/8	101/210 101/245 101/280	

复习题

- 19.23 为什么 GenericMatrix 类中的 add、multiple 以及 zero 方法定义为抽象的?
- 19.24 IntegerMatrix 类中 add、multiple 以及 zero 方法是如何实现的?
- 19.25 RationalMatrix 类中 add、multiple 以及 zero 方法是如何实现的?
- 19.26 如果 printResult 方法如下定义, 将会报什么错?

```

public static void printResult(
    E[][] m1, E[][] m2, E[][] m3, char op)

```

关键术语

actual concrete type (实际具体类型)	generic instantiation (泛型实例化)
bounded generic type (受限泛型类型)	lower-bound wildcard(<? super E>) (下限通配)
bounded wildcard(<? extends E>) (受限通配)	raw type (原始类型)
formal generic type (形式泛型类型)	unbounded wildcard(<?>) (非受限通配)

本章小结

1. 泛型具有参数化类型的能力。可以定义使用泛型类型的类或方法, 编译器会用具体类型来替换泛型类型。
2. 泛型的主要优势是能够在编译时而不是运行时检测错误。
3. 泛型类或方法允许指定这个类或方法可以带有的对象类型。如果试图使用带有不兼容对象的类或方法, 编译器会检测出这个错误。
4. 定义在类、接口或者静态方法中的泛型称为形式泛型类型, 随后可以用一个实际具体类型来替换它。替换泛型类型的过程称为泛型实例化。
5. 不使用类型参数的泛型类称为原始类型, 例如 ArrayList。使用原始类型是为了向后兼容 Java 较早的版本。
6. 通配泛型类型有三种形式: ?、? extends T 和 ? super T, 这里的 T 代表一个泛型类型。第一种形式 ? 称为非受限通配, 它和 ? extends Object 是一样的。第二种形式 ? extends T 称为受限通配, 代表 T 或者 T 的一个子类型。第三种类型 ? super T 称为下限通配, 表示 T 或者 T 的一个父类型。
7. 使用称为类型消除的方法来实现泛型。编译器使用泛型类型信息来编译代码, 但是随后消除它。因此, 泛型信息在运行时是不可用的。这个方法能够使泛型代码向后兼容使用原始类型的遗留代码。

8. 不能使用泛型类型参数来创建实例。
9. 不能使用泛型类型参数来创建数组。
10. 不能在静态环境中使用类的泛型类型参数。
11. 在异常类中不能使用泛型类型参数。

测试题

回答位于网址 www.cs.armstrong.edu/liang/intro10e/quiz.html 的本章测试题。

编程练习题

- 19.1 (修改程序清单 19-1) 修改程序清单 19-1 中的 `GenericStack` 类, 使用数组而不是 `ArrayList` 来实现它。你应该在给栈添加新元素之前检查数组的大小。如果数组满了, 就创建一个新数组, 该数组是当前数组大小的两倍, 然后将当前数组的元素复制到新数组中。
- 19.2 (使用继承实现 `GenericStack`) 程序清单 19-1 中, `GenericStack` 是使用组合实现的。定义一个新的继承自 `ArrayList` 的栈类。画出 UML 类图, 然后实现 `GenericStack`。编写一个测试程序, 提示用户输入 5 个字符串, 然后以逆序显示它们。
- 19.3 (`ArrayList` 中的不同元素) 编写以下方法, 返回一个新的 `ArrayList`。新的列表中包含来自原列表中的不重复元素。

```
public static <E> ArrayList<E> removeDuplicates(ArrayList<E> list)
```

- 19.4 (泛型线性搜索) 为线性搜索实现以下泛型方法。

```
public static <E extends Comparable<E>>  
    int linearSearch(E[] list, E key)
```

- 19.5 (数组中的最大元素) 实现下面的方法, 返回数组中的最大元素。

```
public static <E extends Comparable<E>> E max(E[] list)
```

- 19.6 (二维数组中的最大元素) 编写一个泛型方法, 返回二维数组中的最大元素。

```
public static <E extends Comparable<E>> E max(E[][] list)
```

- 19.7 (泛型二分查找法) 使用二分查找法实现下面的方法。

```
public static <E extends Comparable<E>>  
    int binarySearch(E[] list, E key)
```

- 19.8 (打乱 `ArrayList`) 编写以下方法, 打乱 `ArrayList`。

```
public static <E> void shuffle(ArrayList<E> list)
```

- 19.9 (对 `ArrayList` 排序) 编写以下方法, 对 `ArrayList` 排序。

```
public static <E extends Comparable<E>>  
    void sort(ArrayList<E> list)
```

- 19.10 (`ArrayList` 中的最大元素) 编写以下方法, 返回 `ArrayList` 中的最大元素。

```
public static <E extends Comparable<E>> E max(ArrayList<E> list)
```

- 19.11 (`ComplexMatrix`) 使用编程练习题 13.17 中所介绍的 `Complex` 类来开发 `ComplexMatrix` 类, 用于执行涉及复数的矩阵运算。`ComplexMatrix` 类继承自 `GenericMatrix` 类并实现 `add`、`multiple` 以及 `zero` 方法。你需要修改 `GenericMatrix` 并将每个出现的 `Number` 替换为 `Object`, 因为 `Complex` 不是 `Number` 的子类。编写一个测试程序, 创建两个矩阵并且调用 `printResult` 方法显示它们相加和相乘的结果。

线性表、栈、队列和优先队列

【】 教学目标

- 探索 Java 合集框架层次结构中接口和类的关系 (20.2 节)。
- 使用 Collection 接口中定义的通用方法来操作合集 (20.2 节)。
- 使用 Iterator 接口来遍历一个合集中和元素 (20.3 节)。
- 使用 foreach 循环遍历合集中的元素 (20.3 节)。
- 探索如何使用以及何时使用 ArrayList 或 LinkedList 来存储元素线性表 (20.4 节)。
- 使用 Comparable 接口和 Comparator 接口来比较元素 (20.5 节)。
- 使用 Collections 类中的静态工具方法来排序、查找和打乱线性表, 以及找出合集中的最大元素和最小元素 (20.6 节)。
- 使用 ArrayList 开发多个弹球的应用程序 (20.7 节)。
- 区分 Vector 与 ArrayList, 然后使用 Stack 类创建栈 (20.8 节)。
- 探索 Collection、Queue、LinkedList 以及 PriorityQueue 之间的关系, 然后使用 PriorityQueue 类创建优先队列 (20.9 节)。
- 使用栈编写一个程序, 对表达式求值 (20.10 节)。

20.1 引言

🔑 要点提示: 为一个特定的任务选择最好的数据结构和算法是开发高性能软件的一个关键。

数据结构 (data structure) 是以某种形式将数据组织在一起的合集 (collection)。数据结构不仅存储数据, 还支持访问和处理数据的操作。

在面向对象思想里, 一种数据结构也被认为是一个容器 (container) 或者容器对象 (container object), 它是一个能存储其他对象的对象, 这里的其他对象常称为数据或者元素。定义一种数据结构从本质上讲就是定义一个类。数据结构类应该使用数据域存储数据, 并提供方法支持查找、插入和删除等操作。因此, 创建一个数据结构就是创建这个类的一个实例。然后, 可以使用这个实例上的方法来操作这个数据结构, 例如, 向该数据结构中插入一个元素, 或者从这个数据结构中删除一个元素。

11.11 节已经介绍了 ArrayList 类, 它是一种将元素存储在线性表中的数据结构。Java 还提供了更多能有效地组织和操作数据的数据结构。这些数据结构通常称为 Java 合集框架 (Java Collections Framework)。我们将在本章中介绍线性表 (list)、向量、栈、队列和优先队列的应用, 在下一章中介绍集合 (set) 和映射表 (map)。这些数据结构的实现将在第 24 ~ 27 章中讨论。

20.2 合集

🔑 要点提示: Collection 接口为线性表、向量、栈、队列, 优先队列以及集合定义了共同的操作。

Java 合集框架支持以下两种类型的容器：

- 一种是为了存储一个元素合集，简称为合集（collection）。
- 另一种是为了存储键 / 值对，称为映射表（map）。

映射表是一个用于使用一个键（key）快速搜索一个元素的高效数据结构。我们将在下一章介绍映射表。现在我们将注意力集中在以下合集上。

- Set 用于存储一组不重复的元素。
- List 用于存储一个有序元素合集。
- Stack 用于存储采用后进先出方式处理的对象。
- Queue 用于存储采用先进先出方式处理的对象。
- Priority Queue 用于存储按照优先级顺序处理的对象。

这些合集的通用特性在接口中定义，而实现是在具体类中提供的，如图 20-1 所示。

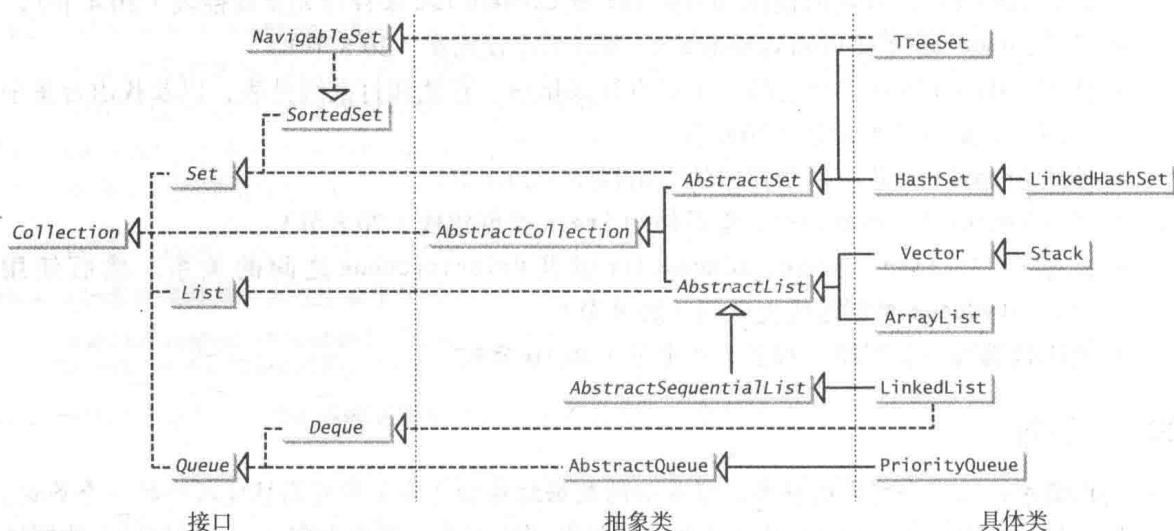


图 20-1 合集是存储对象的容器

注意：在 Java 合集框架中定义的所有接口和类都分组在 `java.util` 包中。

设计指南：Java 合集框架的设计是使用接口、抽象类和具体类的一个很好的例子。用接口定义框架。抽象类提供这个接口的部分实现。具体类用具体的数据结构实现这个接口。提供一个部分实现接口的抽象类对用户编写代码提供了方便。用户可以简单地定义一个具体类继承自抽象类，而无需实现接口中的所有方法。为了方便，提供了如 `AbstractCollection` 这样的抽象类。因为这个原因，这些抽象类被称为便利抽象类（convenience abstract class）。

`Collection` 接口是处理对象合集的根接口。它的公共方法在图 20-2 中列出。`AbstractCollection` 类提供 `Collection` 接口的部分实现。除了 `add`、`size` 和 `iterator` 方法之外，它实现了 `Collection` 接口中的其他所有方法。`add`、`size` 和 `iterator` 等方法在合适的子类中实现。

`Collection` 接口提供了在集合中添加与删除元素的基本操作。`add` 方法给合集添加一个元素。`addAll` 方法把指定集合中的所有元素添加到这个合集中。`remove` 方法从集合中删除一个元素。`removeAll` 方法从这个集合中删除指定集合中的所有元素。`retainAll` 方法保留既出现在这个合集中也出现在指定集合中的元素。所有这些方法都返回 `boolean` 值。如果执

行方法改变了这个合集，那么返回值为 `true`。`clear()` 方法简单移除合集中的所有元素。

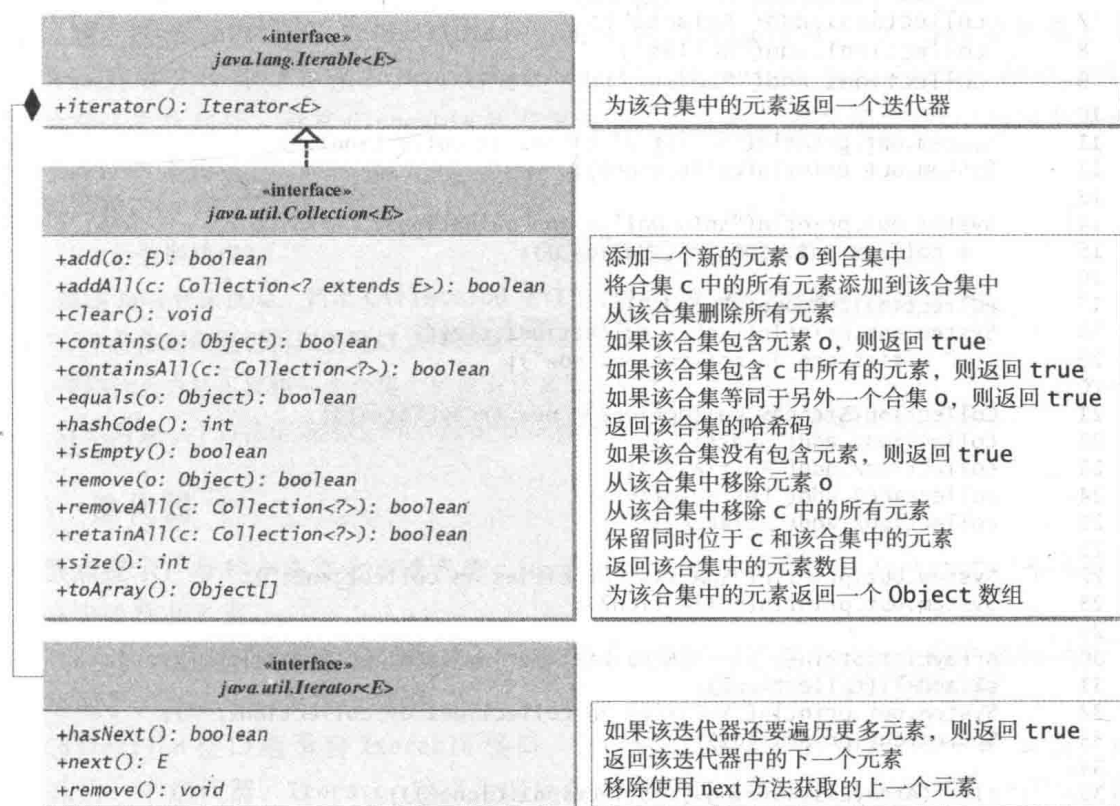


图 20-2 Collection 接口包含了处理合集中元素的方法，并且可以得到一个迭代器对象用于遍历合集中的元素

【注意】 方法 `addAll`、`removeAll`、`retainAll` 类似于集合上的并、差、交运算。

Collection 接口提供了多种查询操作。方法 `size` 返回合集中元素的个数。方法 `contains` 检测合集中是否包含指定的元素。方法 `containsAll` 检测这个合集是否包含指定合集中的所有元素。如果合集为空，方法 `isEmpty` 返回 `true`。

Collection 接口提供的 `toArray()` 方法返回一个合集的数组表示。

【设计指南】 Collection 接口中的有些方法是不能在具体子类中实现的。在这种情况下，这些方法会抛出异常 `java.lang.UnsupportedOperationException`，它是 `RuntimeException` 异常类的一个子类。这样设计很好，可以在自己的项目中使用。如果一个方法在子类中没有意义，可以按如下方式实现它：

```
public void someMethod() {
    throw new UnsupportedOperationException
        ("Method not supported");
}
```

程序清单 20-1 给出了一个使用定义在 Collection 接口中方法的示例。

程序清单 20-1 TestCollection.java

```
1 import java.util.*;
2
3 public class TestCollection {
4     public static void main(String[] args) {
```

```

5     ArrayList<String> collection1 = new ArrayList<>();
6     collection1.add("New York");
7     collection1.add("Atlanta");
8     collection1.add("Dallas");
9     collection1.add("Madison");
10
11     System.out.println("A list of cities in collection1:");
12     System.out.println(collection1);
13
14     System.out.println("\nIs Dallas in collection1? "
15         + collection1.contains("Dallas"));
16
17     collection1.remove("Dallas");
18     System.out.println("\n" + collection1.size() +
19         " cities are in collection1 now");
20
21     Collection<String> collection2 = new ArrayList<>();
22     collection2.add("Seattle");
23     collection2.add("Portland");
24     collection2.add("Los Angeles");
25     collection2.add("Atlanta");
26
27     System.out.println("\nA list of cities in collection2:");
28     System.out.println(collection2);
29
30     ArrayList<String> c1 = (ArrayList<String>)(collection1.clone());
31     c1.addAll(collection2);
32     System.out.println("\nCities in collection1 or collection2: ");
33     System.out.println(c1);
34
35     c1 = (ArrayList<String>)(collection1.clone());
36     c1.retainAll(collection2);
37     System.out.print("\nCities in collection1 and collection2: ");
38     System.out.println(c1);
39
40     c1 = (ArrayList<String>)(collection1.clone());
41     c1.removeAll(collection2);
42     System.out.print("\nCities in collection1, but not in 2: ");
43     System.out.println(c1);
44 }
45 }

```

```

A list of cities in collection1:
[New York, Atlanta, Dallas, Madison]
Is Dallas in collection1? true
3 cities are in collection1 now
A list of cities in collection2:
[Seattle, Portland, Los Angeles, Atlanta]
Cities in collection1 or collection2:
[New York, Atlanta, Madison, Seattle, Portland, Los Angeles, Atlanta]
Cities in collection1 and collection2: [Atlanta]
Cities in collection1, but not in 2: [New York, Madison]

```

程序使用 `ArrayList` 创建了一个具体的合集对象 (第 5 行), 然后调用 `Collection` 接口的 `contains` 方法 (第 15 行)、`remove` 方法 (第 17 行)、`size` 方法 (第 18 行)、`addAll` 方法 (第 31 行)、`retainAll` 方法 (第 36 行) 以及 `removeAll` 方法 (第 41 行)。

对于该例子来说, 我们使用了 `ArrayList`。你可以使用 `Collection` 的任意具体类, 如 `HashSet`、`LinkedList`、`Vector` 以及 `Stack` 替代 `ArrayList` 来测试这些定义在 `Collection` 接口中的方法。

程序创建了一个数组线性表的副本（第 30、35、40 行）。这样做的目的是保持原数组不被改变，而使用它的副本来执行 `addAll`、`retainAll` 以及 `removeAll` 操作。

注意：除开 `java.util.PriorityQueue` 没有实现 `Cloneable` 接口外，Java 合集框架中的其他所有具体类都实现了 `java.lang.Cloneable` 和 `java.io.Serializable` 接口。因此，除开优先队列外，所有 `Cloneable` 的实例都是可克隆的；并且所有 `Cloneable` 的实例都是可序列化的。

复习题

- 20.1 什么是数据结构？
- 20.2 描述 Java 合集框架。列出 `Collection` 接口下面的接口、便利抽象类以及具体类。
- 20.3 一个合集对象是否可以克隆以及序列化？
- 20.4 使用什么方法可以将一个合集中的所有元素添加到另一个合集中？
- 20.5 什么时候一个方法应该抛出 `UnsupportedOperationException` 异常？

20.3 迭代器

要点提示：每种合集都是可迭代的（`Iterable`）。可以获得集合的 `Iterator` 对象来遍历合集中的所有元素。

`Iterator` 是一种经典的设计模式，用于在不需要暴露数据是如何保存在数据结构的细节的情况下，来遍历一个数据结构。

`Collection` 接口继承自 `Iterable` 接口。`Iterable` 接口中定义了 `iterator` 方法，该方法会返回一个迭代器。`Iterator` 接口为遍历各种类型的合集中的元素提供了一种统一的方法。`Iterable` 接口中的 `iterator()` 方法返回一个 `Iterator` 的实例，如图 20-2 所示，它使用 `next()` 方法提供了对合集中元素的顺序访问。还可以使用 `hasNext()` 方法来检测迭代器中是否还有更多的元素，以及 `remove()` 方法来移除迭代器返回的最后一个元素。

程序清单 20-2 TestIterator.java

```
1 import java.util.*;
2
3 public class TestIterator {
4     public static void main(String[] args) {
5         Collection<String> collection = new ArrayList<>();
6         collection.add("New York");
7         collection.add("Atlanta");
8         collection.add("Dallas");
9         collection.add("Madison");
10
11         Iterator<String> iterator = collection.iterator();
12         while (iterator.hasNext()) {
13             System.out.print(iterator.next().toUpperCase() + " ");
14         }
15         System.out.println();
16     }
17 }
```

NEW YORK ATLANTA DALLAS MADISON

程序使用 `ArrayList`（第 5 行）创建一个具体的合集对象，然后添加 4 个字符串到线性表中（第 6 ~ 9 行）。程序然后获得合集的一个迭代器（第 11 行），并使用该迭代器来遍历线性表中的所有字符串，然后以大写方式来显示该字符串（第 12 ~ 14 行）。

技巧：可以使用 `foreach` 循环来简化第 11 ~ 14 行的代码，而不使用迭代器，如下所示：

```
for (String element: collection)
    System.out.print(element.toUpperCase() + " ");
```

该循环可以读为“对合集集中的每个元素，做以下事情。”`foreach` 循环可以用于数组（见 7.2.7 节），也可以用于 `Iterable` 的任何实例。

复习题

- 20.6 如何获得一个合集对象的迭代器？
- 20.7 使用什么方法来从迭代器得到合集中的一个元素？
- 20.8 可以使用 `foreach` 循环来遍历任何 `Collection` 实例中的元素吗？
- 20.9 使用 `foreach` 循环来遍历一个合集中的所有元素时，需要使用迭代器中的 `next()` 或者 `hasNext()` 方法吗？

20.4 线性表

要点提示：`List` 接口继承自 `Collection` 接口，定义了一个用于顺序存储元素的合集。可以使用它的两个具体类 `ArrayList` 或者 `LinkedList` 来创建一个线性表（list）。

前面小节中我们使用了 `ArrayList` 来测试 `Collection` 接口中的方法。现在我们将更深入地来考察 `ArrayList`。本节中我们还将介绍另外一种有用的线性表——`LinkedList`。

20.4.1 List 接口中的通用方法

`ArrayList` 和 `LinkedList` 定义在 `List` 接口下。`List` 接口继承 `Collection` 接口，定义了一个允许重复的有序合集。`List` 接口增加了面向位置的操作，并且增加了一个能够双向遍历线性表的新线性表迭代器。`List` 接口中的方法如图 20-3 所示。

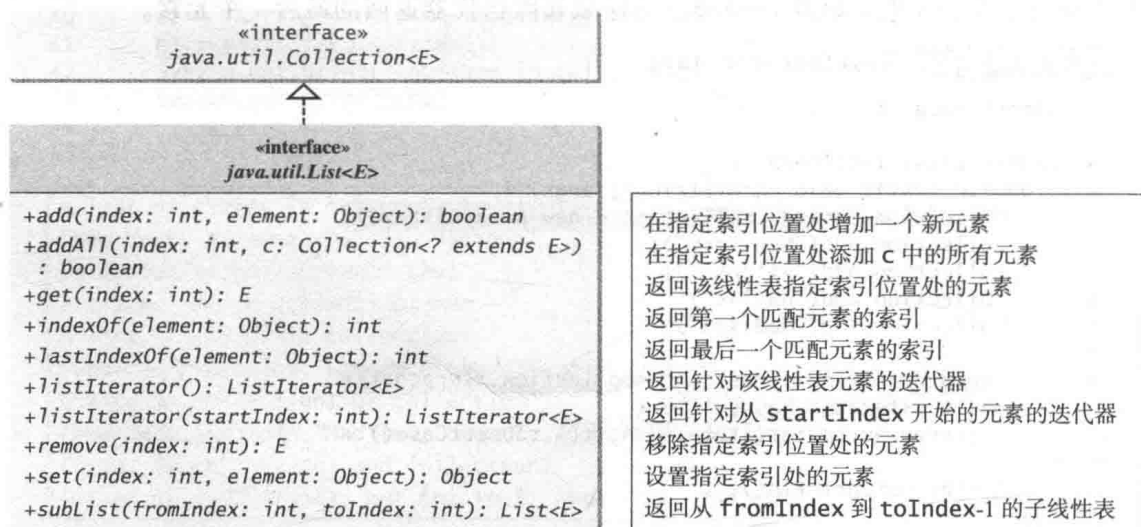


图 20-3 List 接口顺序存储元素并允许元素重复

方法 `add(index, element)` 用于在指定下标处插入一个元素，而方法 `addAll(index, collection)` 用于在指定下标处插入一个元素的合集。方法 `remove(index)` 用于从线性表中删除指定下标处的元素。使用方法 `set(index, element)` 可以在指定下标处设置一个新

元素。

方法 `indexOf(element)` 用于获取指定元素在线性表中第一次出现时的下标，而方法 `lastIndexOf(element)` 用于获取指定元素在线性表中最后一次出现时的下标。使用方法 `subList(fromIndex, toIndex)` 可以获得一个子线性表。

方法 `listIterator()` 或 `listIterator(startIndex)` 都会返回 `ListIterator` 的一个实例。`ListIterator` 接口继承了 `Iterator` 接口，以增加对线性表的双向遍历能力。`ListIterator` 接口中的方法如图 20-4 所示。

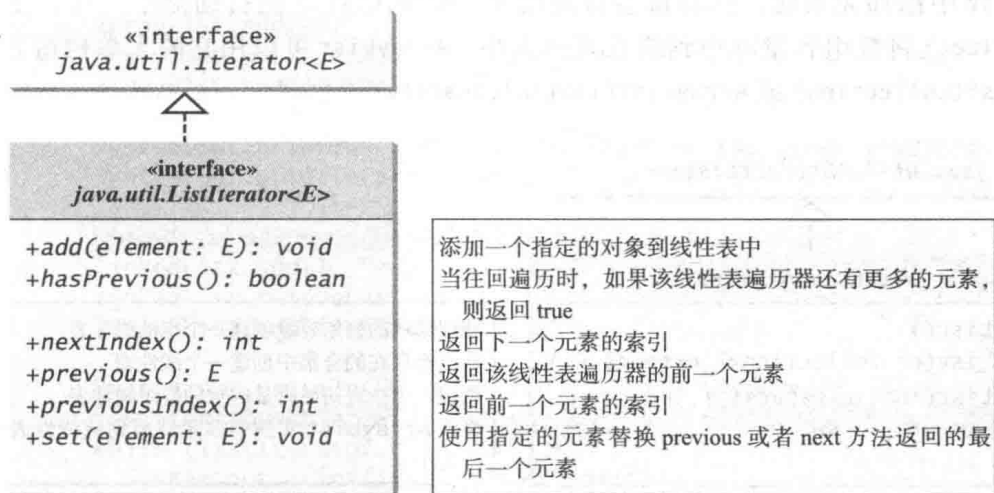


图 20-4 `ListIterator` 接口可以双向遍历线性表

方法 `add(element)` 用于将指定元素插入线性表中。如果 `Iterator` 接口中定义的 `next()` 方法的返回值非空，该元素将被插入到 `next()` 方法返回的元素之前；如果 `previous()` 方法的返回值非空，该元素将被插入到 `previous()` 方法返回的元素之后。如果线性表中没有元素，这个新元素即成为线性表中唯一的元素。`set(element)` 方法用于将 `next` 方法或 `previous` 方法返回的最后一个元素替换为指定元素。

在 `Iterator` 接口中定义的方法 `hasNext()` 用于检测迭代器向前遍历时是否还有元素，而方法 `hasPrevious()` 用于检测迭代器往回遍历时是否还有元素。

在 `Iterator` 接口中定义的方法 `next()` 返回迭代器中的下一个元素，而方法 `previous()` 返回迭代器中的前一个元素。方法 `nextIndex()` 返回迭代器中下一个元素的下标，而方法 `previousIndex()` 返回迭代器中前一个元素的下标。

`AbstractList` 类提供了 `List` 接口的部分实现。`AbstractSequentialList` 类扩展了 `AbstractList` 类，以提供对链表的支持。

20.4.2 数组线性表类 `ArrayList` 和链表类 `LinkedList`

数组线性表类 `ArrayList` 和链表类 `LinkedList` 是实现 `List` 接口的两个具体类。`ArrayList` 用数组存储元素，这个数组是动态创建的。如果元素个数超过了数组的容量，就创建一个更大的新数组，并将当前数组中的所有元素都复制到新数组中。`LinkedList` 在一个链表 (linked list) 中存储元素。要选用这两种类中的哪一个依赖于特定需求。如果需要通过下标随机访问元素，而不会在线性表起始位置插入或删除元素，那么 `ArrayList` 提供了最高

效率的合集。但是，如果应用程序需要在线性表的起始位置上插入或删除元素，就应该选择 `LinkedList` 类。线性表的大小是可以动态增大或减小的。然而数组一旦被创建，它的大小就是固定的。如果应用程序不需要在线性表中插入或删除元素，那么数组是效率最高的数据结构。

`ArrayList` 使用可变大小的数组实现 `List` 接口。它还提供一些方法，用于管理存储线性表的内部数组的大小，如图 20-5 所示。每个 `ArrayList` 实例都有它的容量，这个容量是指存储线性表中元素的数组的大小。它一定不小于所存储的线性表的大小。向 `ArrayList` 中添加元素时，其容量会自动增大。`ArrayList` 不能自动减小。可以使用方法 `trimToSize()` 将数组容量减小到线性表的大小。`ArrayList` 可以用它的无参构造方法——`ArrayList(Collection)` 或 `ArrayList(initialCapacity)` 来创建。

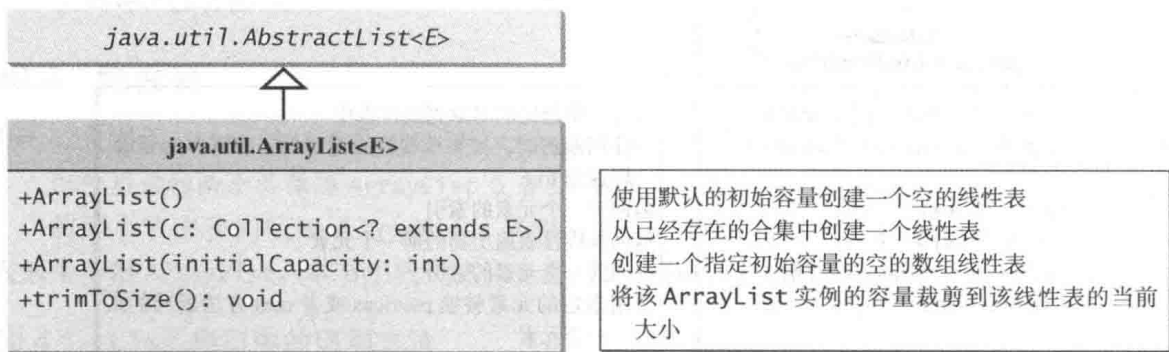


图 20-5 `ArrayList` 使用数组实现 `List`

`LinkedList` 使用链表实现 `List` 接口。除了实现 `List` 接口外，这个类还提供从线性表两端提取、插入和删除元素的方法，如图 20-6 所示。`LinkedList` 可以用它的无参构造方法或 `LinkedList(Collection)` 来创建。

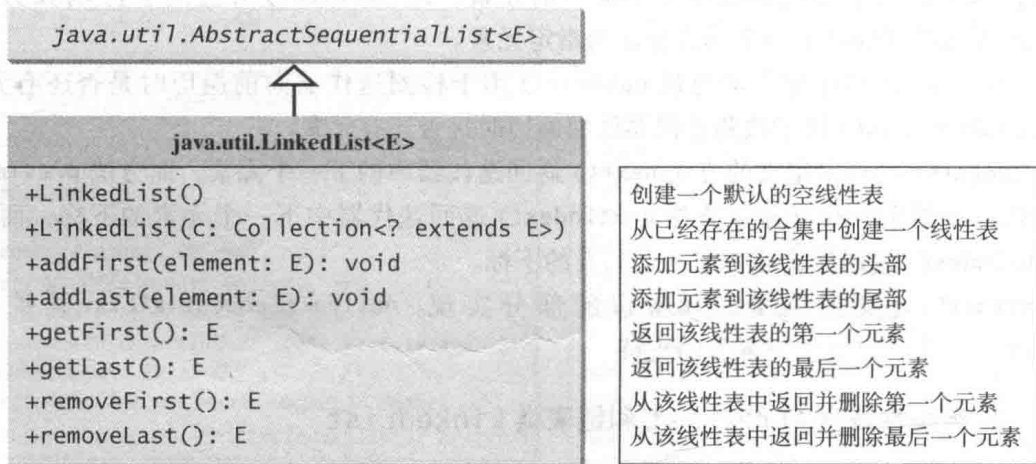


图 20-6 `LinkedList` 提供从线性表两端添加和插入元素的方法

程序清单 20-3 给出一个程序，创建一个用数字填充的数组线性表，并且将新元素插入到线性表的指定位置。本例还从数组线性表创建了一个链表，并且向该链表中插入和删除元素。最后，这个例子分别向前、向后遍历该链表。

程序清单 20-3 TestArrayAndLinkedList.java

```
1 import java.util.*;
2
3 public class TestArrayAndLinkedList {
4     public static void main(String[] args) {
5         List<Integer> arrayList = new ArrayList<>();
6         arrayList.add(1); // 1 is autoboxed to new Integer(1)
7         arrayList.add(2);
8         arrayList.add(3);
9         arrayList.add(1);
10        arrayList.add(4);
11        arrayList.add(0, 10);
12        arrayList.add(3, 30);
13
14        System.out.println("A list of integers in the array list:");
15        System.out.println(arrayList);
16
17        LinkedList<Object> linkedList = new LinkedList<>(arrayList);
18        linkedList.add(1, "red");
19        linkedList.removeLast();
20        linkedList.addFirst("green");
21
22        System.out.println("Display the linked list forward:");
23        ListIterator<Object> listIterator = linkedList.listIterator();
24        while (listIterator.hasNext()) {
25            System.out.print(listIterator.next() + " ");
26        }
27        System.out.println();
28
29        System.out.println("Display the linked list backward:");
30        listIterator = linkedList.listIterator(linkedList.size());
31        while (listIterator.hasPrevious()) {
32            System.out.print(listIterator.previous() + " ");
33        }
34    }
35 }
```

```
A list of integers in the array list:
[10, 1, 2, 30, 3, 1, 4]
Display the linked list forward:
green 10 red 1 2 30 3 1
Display the linked list backward:
1 3 30 2 1 red 10 green
```

线性表可以存储相同的元素。整数 1 就在线性表中存储了两次（第 6 和 9 行）。ArrayList 和 LinkedList 的操作类似，它们最主要的不同体现在内部实现上，内部实现会影响到它们的性能。ArrayList 获取元素的效率比较高；若在线性表的起始位置插入和删除元素，那么 LinkedList 的效率会高一些。两种线性表在中间或者末尾位置上插入和删除元素方面具有同样的性能。

链表可以使用 get(i) 方法，但这是一个耗时的操作。不要使用它来遍历线性表中的所有元素，如 a 所示。应该使用一个迭代器，如 b 所示。注意 foreach 循环隐式地使用了迭代器。当在第 24 章中学习如何实现一个链表的时候，你将知道原因。

```
for (int i = 0; i < list.size(); i++) {
    process list.get(i);
}
```

a) 非常低效

```
for (listElementType s: list) {
    process s;
}
```

b) 高效的

提示：为了从可变长参数表中创建线性表，Java 提供了静态的 `asList` 方法。这样，就可以使用下面的代码创建一个字符串线性表和一个整数线性表：

```
List<String> list1 = Arrays.asList("red", "green", "blue");
List<Integer> list2 = Arrays.asList(10, 20, 30, 40, 50);
```

复习题

- 20.10 如何向线性表中添加元素和从线性表中删除元素？如何从两个方向遍历线性表？
- 20.11 假设 `list1` 是一个包含字符串 `red`、`yellow`、`green` 的线性表，`list2` 是一个包含字符串 `red`、`yellow`、`blue` 的线性表，回答下面的问题：
- 执行完 `list1.addAll(list2)` 方法之后，线性表 `list1` 和 `list2` 分别变成了什么？
 - 执行完 `list1.add(list2)` 方法之后，线性表 `list1` 和 `list2` 分别变成了什么？
 - 执行完 `list1.removeAll(list2)` 方法之后，线性表 `list1` 和 `list2` 分别变成了什么？
 - 执行完 `list1.remove(list2)` 方法之后，线性表 `list1` 和 `list2` 分别变成了什么？
 - 执行完 `list1.retainAll(list2)` 方法之后，线性表 `list1` 和 `list2` 分别变成了什么？
 - 执行完 `list1.clear()` 方法之后，线性表 `list1` 变成了什么？
- 20.12 `ArrayList` 与 `LinkedList` 之间的区别是什么？应该使用哪种线性表在一个线性表的起始位置插入和删除元素。
- 20.13 `LinkedList` 是否包含 `ArrayList` 中的所有方法？哪些方法在 `LinkedList` 中有，但在 `ArrayList` 中却没有？
- 20.14 如何从一个对象数组创建一个线性表？

20.5 Comparator 接口

要点提示：`Comparator` 可以用于比较没有实现 `Comparable` 的类的对象。

你已经学习了如何使用 `Comparable` 接口来比较元素（13.6 节中介绍）。Java API 的一些类，比如 `String`、`Date`、`Calendar`、`BigInteger`、`BigDecimal` 以及所有基本类型的数字包装类都实现了 `Comparable` 接口。`Comparable` 接口定义了 `compareTo` 方法，用于比较实现了 `Comparable` 接口的同一个类的两个元素。

如果元素的类没有实现 `Comparable` 接口又将如何呢？这些元素可以比较么？可以定义一个比较器（`comparator`）来比较不同类的元素。要做到这一点，需要创建一个实现 `java.util.Comparator<T>` 接口的类并重写它的 `compare` 方法。

```
public int compare(T element1, T element2)
```

如果 `element1` 小于 `element2`，就返回一个负值；如果 `element1` 大于 `element2`，就返回一个正值；若两者相等，则返回 0。

表 13.2 节介绍了 `GeometricObject` 类。`GeometricObject` 类没有实现 `Comparable` 接口。为了比较 `GeometricObject` 类的对象，可以定义一个比较器类，如程序清单 20-4 所示。

程序清单 20-4 GeometricObjectComparator.java

```
1 import java.util.Comparator;
```

```

2
3 public class GeometricObjectComparator
4     implements Comparator<GeometricObject>, java.io.Serializable {
5     public int compare(GeometricObject o1, GeometricObject o2) {
6         double area1 = o1.getArea();
7         double area2 = o2.getArea();
8
9         if (area1 < area2)
10             return -1;
11         else if (area1 == area2)
12             return 0;
13         else
14             return 1;
15     }
16 }

```

第4行实现了 `Comparator<GeometricObject>`，第5行通过覆盖 `compare` 方法来比较两个几何对象。比较器类也实现了 `Serializable` 接口。通常对于比较器来说，实现 `Serializable` 是一个好主意，因为它们可以被用作可序列化数据结构的排序方法。为了使数据结构能够成功序列化，比较器（如果提供）必须实现 `Serializable` 接口。

程序清单 20-5 给出了一个方法，返回两个几何对象中较大的那个。两个对象使用 `GeometricObjectComparator` 进行比较。

程序清单 20-5 TestComparator.java

```

1 import java.util.Comparator;
2
3 public class TestComparator {
4     public static void main(String[] args) {
5         GeometricObject g1 = new Rectangle(5, 5);
6         GeometricObject g2 = new Circle(5);
7
8         GeometricObject g =
9             max(g1, g2, new GeometricObjectComparator());
10
11         System.out.println("The area of the larger object is " +
12             g.getArea());
13     }
14
15     public static GeometricObject max(GeometricObject g1,
16         GeometricObject g2, Comparator<GeometricObject> c) {
17         if (c.compare(g1, g2) > 0)
18             return g1;
19         else
20             return g2;
21     }
22 }

```

```
The area of the larger object is 78.53981633974483
```

程序在第5~6行创建了一个 `Rectangle` 对象和一个 `Circle` 对象（`Rectangle` 类和 `Circle` 类在第13.2节中定义）。它们都是 `GeometricObject` 的子类。程序调用 `max` 方法得到具有较大面积的几何对象（第8~9行）。

`GeometricObjectComparator` 被创建并且传递给 `max` 方法（第9行），程序第17行中 `max` 方法使用了比较器来比较几何对象。

注意：`Comparable` 用于比较实现 `Comparable` 的类的对象；`Comparator` 用于比较没有实现 `Comparable` 的类的对象。

使用 `Comparable` 接口来比较元素称为使用自然顺序（`natural order`）进行比较，使用

Comparator 接口来比较元素被称为使用比较器来进行比较。

复习题

20.15 Comparable 接口与 Comparator 接口之间有什么不同之处？它们分别属于哪一个包？

20.16 如何定义一个实现 Comparable 接口的类 A？类 A 的两个实例可以比较吗？如何定义一个实现了 Comparator 接口的类 B，并且重写 compare 方法来比较类 B1 的对象？如何调用 sort 方法来对类 B1 的对象线性表进行排序？

20.6 线性表和合集的静态方法

要点提示： Collections 类包含了执行合集和线性表中通用操作的静态方法。

11.12 节中介绍了 Collections 类中针对数组线性表的一些静态方法。Collections 类包含用于线性表的 sort、binarySearch、reverse、shuffle、copy 和 fill 方法，以及用于合集的 max、min、disjoint 和 frequency 方法，如图 20-7 所示。

java.util.Collections	
线性表	+sort(list: List): void
	+sort(list: List, c: Comparator): void
	+binarySearch(list: List, key: Object): int
	+binarySearch(list: List, key: Object, c: Comparator): int
	+reverse(list: List): void
	+reverseOrder(): Comparator
	+shuffle(list: List): void
	+shuffle(list: List, rnd: Random): void
	+copy(des: List, src: List): void
	+nCopies(n: int, o: Object): List
合集	+fill(list: List, o: Object): void
	+max(c: Collection): Object
	+max(c: Collection, c: Comparator): Object
	+min(c: Collection): Object
	+min(c: Collection, c: Comparator): Object
	+disjoint(c1: Collection, c2: Collection): boolean
	+frequency(c: Collection, o: Object): int

对指定的线性表进行排序
使用比较器对指定的线性表进行排序
采用二分查找来找到排好序的线性表中的键值
使用比较器，采用二分查找来找到排好序的线性表中的键值
对指定的线性表进行逆序排序
返回一个逆序排序的比较器
随机打乱指定的线性表
使用一个随机对象打乱指定的线性表
复制源线性表到目标线性表中
返回一个由 n 个对象副本组成的线性表
使用对象填充线性表
返回合集中的 max 对象
使用比较器返回 max 对象
返回合集中的 min 对象
使用比较器返回 min 对象
如果 c1 和 c2 没有共同的元素，则返回真
返回合集中指定元素的出现次数

图 20-7 Collections 类包含操作线性表和合集的静态方法

可以使用 Comparable 接口中的 compareTo 方法，对线性表中的可比较的元素以自然顺序排序。也可以指定比较器来对元素排序。例如，下面的代码就是对线性表中的字符串排序：

```
List<String> list = Arrays.asList("red", "green", "blue");
Collections.sort(list);
System.out.println(list);
```

输出为 [blue, green, red]。

上面的代码以升序对线性表排序。要以降序排列，可以简单地使用 Collections.reverseOrder() 方法返回一个 Comparator 对象，该方法以逆序排列元素。例如，下面的代码就是对字符串线性表以降序排列：

```
List<String> list = Arrays.asList("yellow", "red", "green", "blue");
Collections.sort(list, Collections.reverseOrder());
System.out.println(list);
```

输出为 [yellow, red, green, blue]。

使用 `binarySearch` 方法可以在线性表中查找一个键值。这个线性表必须提前以升序排列好。如果这个键值没有在线性表中，那么这个方法就会返回 $(-insertion\ point + 1)$ 。回忆一下，如果存在一个条目，插入点就是条目在线性表中的位置。例如，下面的代码在一个整数线性表和一个字符串线性表中查找键值：

```
List<Integer> list1 =
    Arrays.asList(2, 4, 7, 10, 11, 45, 50, 59, 60, 66);
System.out.println("(1) Index: " + Collections.binarySearch(list1, 7));
System.out.println("(2) Index: " + Collections.binarySearch(list1, 9));

List<String> list2 = Arrays.asList("blue", "green", "red");
System.out.println("(3) Index: " +
    Collections.binarySearch(list2, "red"));
System.out.println("(4) Index: " +
    Collections.binarySearch(list2, "cyan"));
```

上面代码的输出为

```
(1) Index: 2
(2) Index: -4
(3) Index: 2
(4) Index: -2
```

可以使用 `reverse` 方法将线性表中的元素以逆序排列。例如，下面的代码显示 [blue, green, red, yellow]：

```
List<String> list = Arrays.asList("yellow", "red", "green", "blue");
Collections.reverse(list);
System.out.println(list);
```

可以使用 `shuffle(List)` 方法对线性表中的元素进行随机重新排序。例如，下面的代码打乱 `list` 中的元素：

```
List<String> list = Arrays.asList("yellow", "red", "green", "blue");
Collections.shuffle(list);
System.out.println(list);
```

也可以使用 `shuffle(List, Random)` 方法以一个指定的 `Random` 对象对线性表中的元素随机重新排序。要产生一个和原始线性表拥有相同元素序列的线性表，使用指定的 `Random` 对象是很有用的。例如，下面的代码打乱 `list` 中的元素：

```
List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");
List<String> list2 = Arrays.asList("yellow", "red", "green", "blue");
Collections.shuffle(list1, new Random(20));
Collections.shuffle(list2, new Random(20));
System.out.println(list1);
System.out.println(list2);
```

你将看到 `list1` 和 `list2` 在打乱之前和之后拥有相同的元素序列。

可以使用 `copy(dest, src)` 方法将源线性表中的所有元素以同样的下标复制到目标线性表中。目标线性表必须和源线性表等长。如果源线性表的长度大于目标线性表，那么，目标线性表中的剩余元素不会受到影响。例如，下面的代码将 `list2` 复制到 `list1` 中：

```
List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");
List<String> list2 = Arrays.asList("white", "black");
```

```
Collections.copy(list1, list2);  
System.out.println(list1);
```

list1 的输出是 [white,black,green,blue]。copy 方法执行的是浅复制。复制的只是源线性表中元素的引用。

可以使用方法 `nCopies(int n, object o)` 创建一个包含指定对象的 `n` 个副本的不可变线性表。例如，下面的代码用 5 个 `Calendar` 对象创建一个线性表：

```
List<GregorianCalendar> list1 = Collections.nCopies  
(5, new GregorianCalendar(2005, 0, 1));
```

用 `nCopies` 方法创建的线性表是不可变的，因此，不能在该线性表中添加、删除或更新元素。所有的元素都有相同的引用。

可以使用 `fill(List list, Object o)` 方法来用指定元素替换线性表中的所有元素。例如，下面的代码显示 [black,black,black]：

```
List<String> list = Arrays.asList("red", "green", "blue");  
Collections.fill(list, "black");  
System.out.println(list);
```

可以使用 `max` 和 `min` 方法找出合集中的最大元素和最小元素。集合中的元素必须是可使用 `Comparable` 接口或 `Comparator` 接口比较的。例如，下面的代码显示合集中最大的字符串和最小的字符串：

```
Collection<String> collection = Arrays.asList("red", "green", "blue");  
System.out.println(Collections.max(collection));  
System.out.println(Collections.min(collection));
```

如果两个合集没有相同的元素，那么 `disjoint(collection1, collection2)` 方法返回 `true`。例如，在下面的代码中，`disjoint(collection1, collection2)` 方法返回 `false`，但是 `disjoint(collection1, collection3)` 方法返回 `true`。

```
Collection<String> collection1 = Arrays.asList("red", "cyan");  
Collection<String> collection2 = Arrays.asList("red", "blue");  
Collection<String> collection3 = Arrays.asList("pink", "tan");  
System.out.println(Collections.disjoint(collection1, collection2));  
System.out.println(Collections.disjoint(collection1, collection3));
```

使用 `frequency(collection, element)` 方法可以找出合集中某元素的出现次数。例如，在下面代码中 `frequency(collection, "red")` 返回 2。

```
Collection<String> collection = Arrays.asList("red", "cyan", "red");  
System.out.println(Collections.frequency(collection, "red"));
```

✓ 复习题

20.17 `Collections` 类中的所有方法是否都是静态的？

20.18 下面 `Collections` 类中的哪些静态方法是用于线性表的？哪些是用于合集的？

sort, binarySearch, reverse, shuffle, max, min, disjoint, frequency

20.19 给出下面代码的输出结果：

```
import java.util.*;  
  
public class Test {  
    public static void main(String[] args) {
```

```

List<String> list =
    Arrays.asList("yellow", "red", "green", "blue");
Collections.reverse(list);
System.out.println(list);

List<String> list1 =
    Arrays.asList("yellow", "red", "green", "blue");
List<String> list2 = Arrays.asList("white", "black");
Collections.copy(list1, list2);
System.out.println(list1);


Collection<String> c1 = Arrays.asList("red", "cyan");
Collection<String> c2 = Arrays.asList("red", "blue");
Collection<String> c3 = Arrays.asList("pink", "tan");
System.out.println(Collections.disjoint(c1, c2));
System.out.println(Collections.disjoint(c1, c3));

Collection<String> collection =
    Arrays.asList("red", "cyan", "red");
System.out.println(Collections.frequency(collection, "red"));
}
}

```

- 20.20 使用哪个方法可以对 ArrayList 或 LinkedList 中的元素进行排序？使用哪个方法可以对字符串数组进行排序？
- 20.21 使用哪个方法可以对 ArrayList 或 LinkedList 中的元素进行二分查找？使用哪个方法可以对字符串数组中的元素进行二分查找？
- 20.22 编写一条语句，找出由可比较对象构成的数组中的最大元素。

20.7 示例学习：弹球

 **要点提示：**本节给出一个显示弹球的程序，可以让用户添加和移除球。

15.12 节给出了一个程序显示一个弹球。本节给出一个程序显示多个弹球。可以使用两个按钮来暂停和恢复球的移动，一个滚动条来控制球速，以及 + 和 - 按钮来添加和移除一个球，如图 20-8 所示。

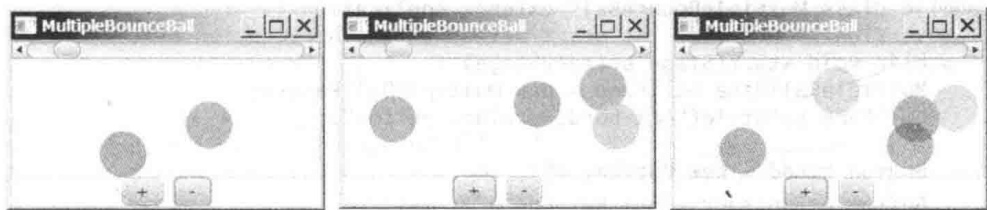


图 20-8 按 + 和 - 按钮来添加和移除球

15.12 节中的例子只需要保存一个球。如何在该例中保存多个球呢？Pane 的 `getChildren()` 方法返回一个 `List<Node>` 的子类 `ObservableList<Node>`，用于存储面板中的结点。该线性表初始为空。当创建一个新的球时，将其添加到线性表的末尾。要移除一个球，只需要简单地将线性表的最后一个移除。

每个球有它的状态： x 坐标、 y 坐标、颜色以及移动的方向。可以定义一个继承自 `javafx.scene.shape.Circle` 的命名为 `Ball` 的类。`Circle` 中已经定义了 x 坐标、 y 坐标以及颜色。当球创建时，它从左上角开始向右下移动。一个随机颜色被赋给一个新的球。

`MultipleBallPane` 类负责显示球，`MultipleBounceBall` 类放置控制组件并且实现控制。

这些类的关系显示在图 20-9 中。程序清单 20-6 给出该程序。

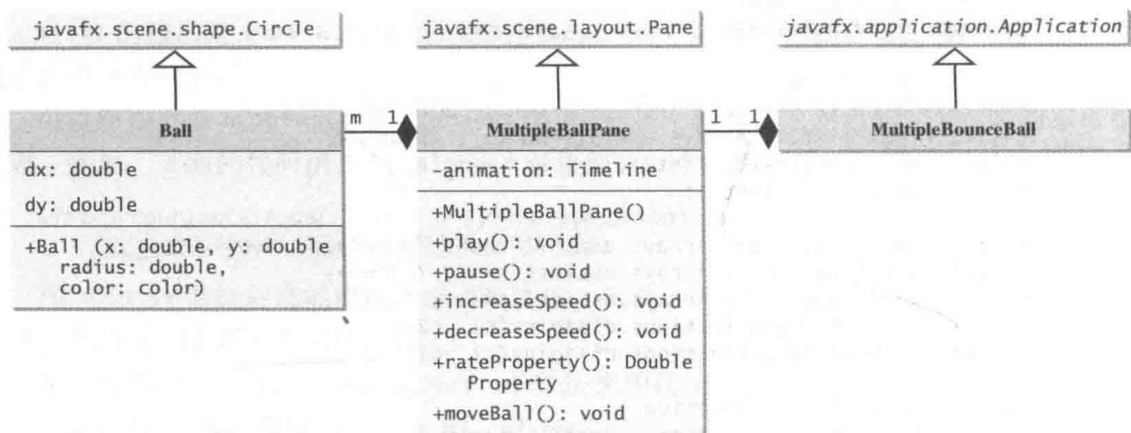


图 20-9 MultipleBounceBall 包含 MultipleBallPane, MultipleBallPane 包含 Ball

程序清单 20-6 MultipleBounceBall.java

```

1  import javafx.animation.KeyFrame;
2  import javafx.animation.Timeline;
3  import javafx.application.Application;
4  import javafx.beans.property.DoubleProperty;
5  import javafx.geometry.Pos;
6  import javafx.scene.Node;
7  import javafx.stage.Stage;
8  import javafx.scene.Scene;
9  import javafx.scene.control.Button;
10 import javafx.scene.control.ScrollBar;
11 import javafx.scene.layout.BorderPane;
12 import javafx.scene.layout.HBox;
13 import javafx.scene.layout.Pane;
14 import javafx.scene.paint.Color;
15 import javafx.scene.shape.Circle;
16 import javafx.util.Duration;
17
18 public class MultipleBounceBall extends Application {
19     @Override // Override the start method in the Application class
20     public void start(Stage primaryStage) {
21         MultipleBallPane ballPane = new MultipleBallPane();
22         ballPane.setStyle("-fx-border-color: yellow");
23
24         Button btAdd = new Button("+");
25         Button btSubtract = new Button("-");
26         HBox hBox = new HBox(10);
27         hBox.getChildren().addAll(btAdd, btSubtract);
28         hBox.setAlignment(Pos.CENTER);
29
30         // Add or remove a ball
31         btAdd.setOnAction(e -> ballPane.add());
32         btSubtract.setOnAction(e -> ballPane.subtract());
33
34         // Pause and resume animation
35         ballPane.setOnMousePressed(e -> ballPane.pause());
36         ballPane.setOnMouseReleased(e -> ballPane.play());
37
38         // Use a scroll bar to control animation speed
39         ScrollBar sbSpeed = new ScrollBar();
40         sbSpeed.setMax(20);
  
```

```

41 sbSpeed.setValue(10);
42 ballPane.rateProperty().bind(sbSpeed.valueProperty());
43
44 BorderPane pane = new BorderPane();
45 pane.setCenter(ballPane);
46 pane.setTop(sbSpeed);
47 pane.setBottom(hBox);
48
49 // Create a scene and place the pane in the stage
50 Scene scene = new Scene(pane, 250, 150);
51 primaryStage.setTitle("MultipleBounceBall"); // Set the stage title
52 primaryStage.setScene(scene); // Place the scene in the stage
53 primaryStage.show(); // Display the stage
54 }
55
56 private class MultipleBallPane extends Pane {
57     private Timeline animation;
58
59     public MultipleBallPane() {
60         // Create an animation for moving the ball
61         animation = new Timeline(
62             new KeyFrame(Duration.millis(50), e -> moveBall()));
63         animation.setCycleCount(Timeline.INDEFINITE);
64         animation.play(); // Start animation
65     }
66
67     public void add() {
68         Color color = new Color(Math.random(),
69             Math.random(), Math.random(), 0.5);
70         getChildren().add(new Ball(30, 30, 20, color));
71     }
72
73     public void subtract() {
74         if (getChildren().size() > 0) {
75             getChildren().remove(getChildren().size() - 1);
76         }
77     }
78
79     public void play() {
80         animation.play();
81     }
82
83     public void pause() {
84         animation.pause();
85     }
86
87     public void increaseSpeed() {
88         animation.setRate(animation.getRate() + 0.1);
89     }
90
91     public void decreaseSpeed() {
92         animation.setRate(
93             animation.getRate() > 0 ? animation.getRate() - 0.1 : 0);
94     }
95
96     public DoubleProperty rateProperty() {
97         return animation.rateProperty();
98     }
99
100     protected void moveBall() {
101         for (Node node: this.getChildren()) {
102             Ball ball = (Ball)node;
103             // Check boundaries
104             if (ball.getCenterX() < ball.getRadius() ||

```



```

105         ball.getCenterX() > getWidth() - ball.getRadius()) {
106             ball.dx *= -1; // Change ball move direction
107         }
108         if (ball.getCenterY() < ball.getRadius() ||
109             ball.getCenterY() > getHeight() - ball.getRadius()) {
110             ball.dy *= -1; // Change ball move direction
111         }
112
113         // Adjust ball position
114         ball.setCenterX(ball.dx + ball.getCenterX());
115         ball.setCenterY(ball.dy + ball.getCenterY());
116     }
117 }
118 }
119
120 class Ball extends Circle {
121     private double dx = 1, dy = 1;
122
123     Ball(double x, double y, double radius, Color color) {
124         super(x, y, radius);
125         setFill(color); // Set ball color
126     }
127 }
128 }

```

`add()` 方法用一个随机颜色创建一个新的球并且将它加入到面板中 (第 70 行)。面板将所有的球存储在一个列表中。`subtract()` 方法移除列表中的最后一个球 (第 75 行)。


当用户单击 + 按钮时, 一个新的球被加入到面板中 (第 31 行)。当用户单击 - 按钮时, 数组列表中的最后一个球被移除 (第 32 行)。

`MultipleBallPane` 中的 `moveBall()` 方法得到面板中列表里面的每个球, 并且调整球的位置 (第 114 ~ 115 行)。

✓ 复习题

- 20.23 对一个面板调用 `pane.getChildren()` 将返回什么值?
- 20.24 如何修改 `MutipleBallApp` 程序中的代码, 使得当按钮被单击的时候移除列表中的第一个球?
- 20.25 如何修改 `MutipleBallApp` 程序中的代码, 从而每个球的半径具有一个 10 和 20 之间的随机值?

20.8 向量类和栈类

 **要点提示:** 在 Java API 中, `Vector` 是 `AbstractList` 的子类, `Stack` 是 `Vector` 的子类。

Java 合集框架是在 Java 2 中引入的。Java 2 之前的版本也支持一些数据结构, 其中就有向量类 `Vector` 与栈类 `Stack`。为了适应 Java 合集框架, Java 2 对这些类进行了重新设计, 但是为了向后兼容, 保留了它们所有的以前形式的方法。

除了包含用于访问和修改向量的同步方法之外, `Vector` 类与 `ArrayList` 是一样的。同步方法用于防止两个或多个线程同时访问和修改某个向量时引起数据损坏。我们将在第 30 章讨论同步问题。对于许多不需要同步的应用程序来说, 使用 `ArrayList` 比使用 `Vector` 效率更高。

`Vector` 类继承了 `AbstractList` 类, 它还包含 Java 2 以前的版本中原始 `Vector` 类中的方法, 如图 20-10 所示。

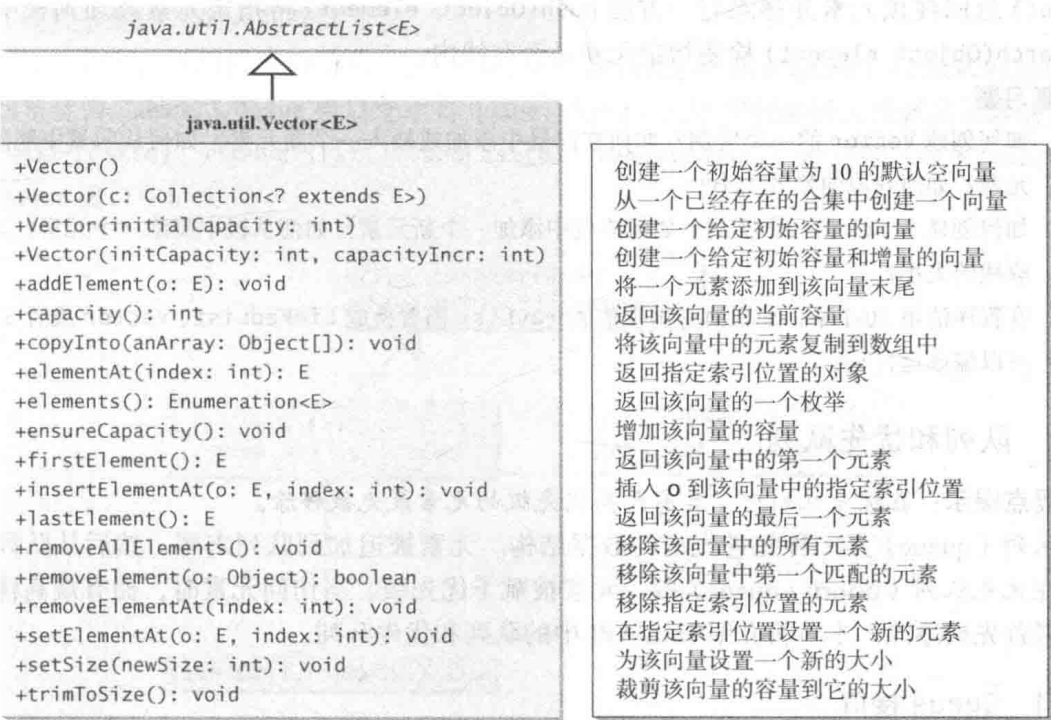


图 20-10 从 Java 2 开始, Vector 类继承了 AbstractList 类, 并保留了原来 Vector 类中的所有方法

图 20-10 中的 UML 图中所列出的 Vector 类中的大多数方法都类似于 List 接口中的方法。这些方法都是在 Java 合集框架之前引入的。例如, addElement(Object element) 方法除了是同步的之外, 它与 add(Object element) 方法是一样的。如果不需要同步, 最好使用 ArrayList 类, 因为它比 Vector 快得多。

注意: 方法 elements() 返回一个 Enumeration 对象 (枚举型对象)。Enumeration 接口是在 Java 2 之前引入的, 已经被 Iterator 接口所取代。

注意: Vector 类被广泛应用于 Java 的遗留代码中, 因为在 Java 2 之前, 它可以实现 Java 可变大小的数组。

在 Java 合集框架中, 栈类 Stack 是作为 Vector 类的扩展来实现的, 如图 20-11 所示。

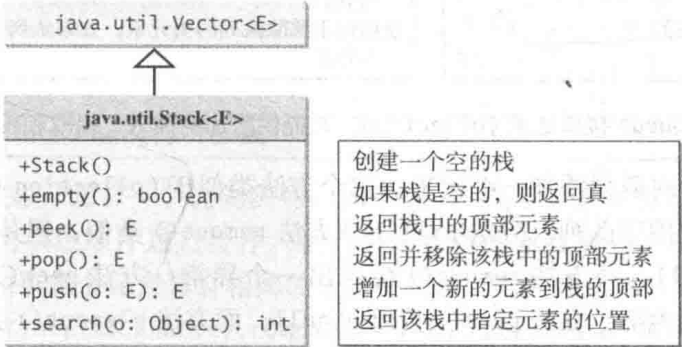


图 20-11 Stack 类继承 Vector, 提供了后进先出的数据结构

Stack 类是在 Java 2 之前引入的。图 20-11 给出的方法在 Java 2 之前已经使用。方法 empty() 与方法 isEmpty() 的功能是一样的。方法 peek() 可以返回栈顶元素而不移除它。方

法 `pop()` 返回栈顶元素并移除它。方法 `push(Object element)` 将指定元素添加到栈中。方法 `search(Object element)` 检测指定元素是否在栈内。

复习题

- 20.26 如何创建 `Vector` 的一个实例？如何在向量中添加或插入一个新元素？如何从向量中删除一个元素？如何获取向量的大小？
- 20.27 如何创建 `Stack` 的一个实例？如何向栈中添加一个新元素？如何从栈中删除一个元素？如何获取栈的大小？
- 20.28 在程序清单 20-1 中，如果所有出现的 `ArrayList` 都替换成 `LinkedList`、`Vector` 或者 `Stack`，可以编译运行吗？

20.9 队列和优先队列

要点提示：在优先队列中，具有最高优先级的元素最先被移除。

队列（queue）是一种先进先出的数据结构。元素被追加到队列末尾，然后从队列头删除。在优先队列（priority queue）中，元素被赋予优先级。当访问元素时，拥有最高优先级的元素首先被删除。本节将介绍 Java API 中的队列和优先队列。

20.9.1 Queue 接口

`Queue` 接口继承自 `java.util.Collection`，加入了插入、提取和检验等操作，如图 20-12 所示。

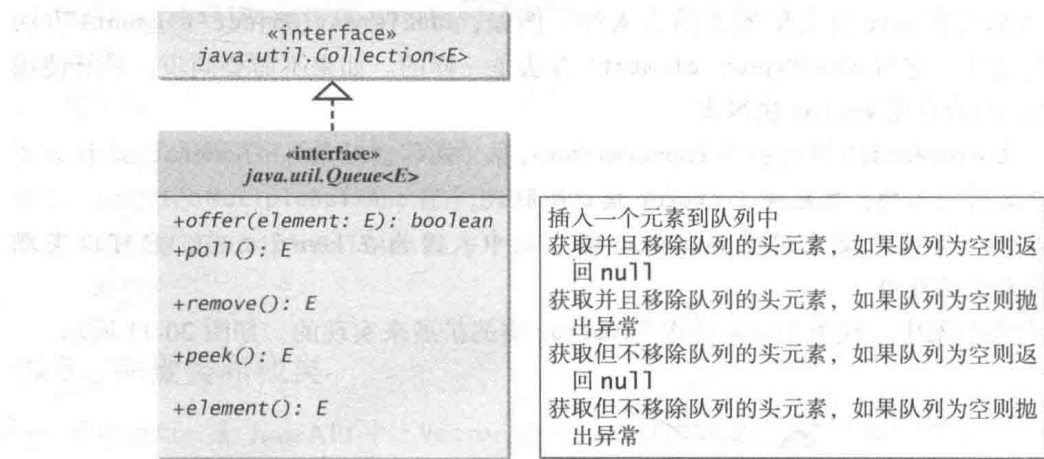


图 20-12 `Queue` 接口继承 `Collection`，并提供附加的插入、提取和检验等操作

方法 `offer` 用于向队列添加一个元素。这个方法类似于 `Collection` 接口中的 `add` 方法，但是 `offer` 方法更适用于队列。方法 `poll()` 和方法 `remove()` 类似，但是如果队列为空，方法 `poll()` 会返回 `null`，而方法 `remove()` 会抛出一个异常。方法 `peek()` 和方法 `element()` 类似，但是如果队列为空，方法 `peek()` 会返回 `null`，而方法 `element()` 会抛出一个异常。

20.9.2 双端队列 `Deque` 和链表 `LinkedList`

`LinkedList` 类实现了 `Deque` 接口，`Deque` 又继承自 `Queue` 接口，如图 20-13 所示。因此，可以使用 `LinkedList` 创建一个队列。`LinkedList` 很适合用于进行队列操作，因为它可以高

效地在线性表的两端插入和移除元素。

Deque 支持在两端插入和删除元素。deque 是 (double-ended queue) 双端队列的简称, 通常的发音为 “deck”。Deque 接口继承自 Queue, 增加了从队列两端插入和删除元素的方法。方法 `addFirst(e)`、`removeFirst()`、`addLast(e)`、`removeLast()`、`getFirst()` 和 `getLast()` 都在 Deque 接口中定义。

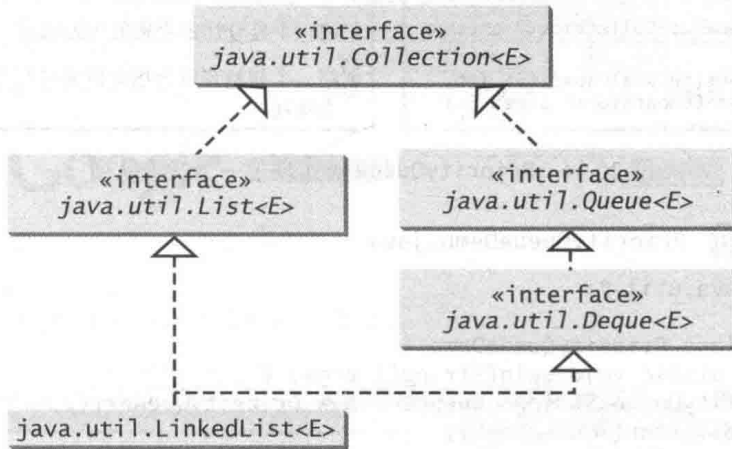


图 20-13 LinkedList 实现了 List 和 Deque

程序清单 20-7 给出一个使用队列存储字符串的例子。程序第 4 行使用 LinkedList 创建一个队列, 第 5 ~ 8 行将 4 个字符串添加到队列中。在 Collection 接口中定义的方法 `size()` 返回队列中的元素数目 (第 10 行)。方法 `remove()` 获取并删除队列头的元素 (第 11 行)。

程序清单 20-7 TestQueue.java

```

1 public class TestQueue {
2     public static void main(String[] args) {
3         java.util.Queue<String> queue = new java.util.LinkedList<>();
4         queue.offer("Oklahoma");
5         queue.offer("Indiana");
6         queue.offer("Georgia");
7         queue.offer("Texas");
8
9         while (queue.size() > 0)
10             System.out.print(queue.remove() + " ");
11     }
12 }

```

Oklahoma Indiana Georgia Texas

PriorityQueue 类实现了一个优先队列, 如图 20-14 所示。默认情况下, 优先队列使用 Comparable 以元素的自然顺序进行排序。拥有最小数值的元素被赋予最高优先级, 因此最先从队列中删除。如果几个元素具有相同的最高优先级, 则任意选择一个。也可以使用构造方法 `PriorityQueue(initialCapacity, comparator)` 中的 Comparator 来指定一个顺序。

程序清单 20-8 给出一个使用优先队列存储字符串的例子。程序第 5 行使用无参构造方法创建字符串优先队列。这个优先队列以字符串的自然顺序进行排序, 这样, 字符串以升序从队列中删除。第 16 ~ 17 行使用从 `Collections.reverseOrder()` 中获得的比较器创建优先队列, 该方法以逆序对元素排序, 因此, 字符串以降序从队列中删除。

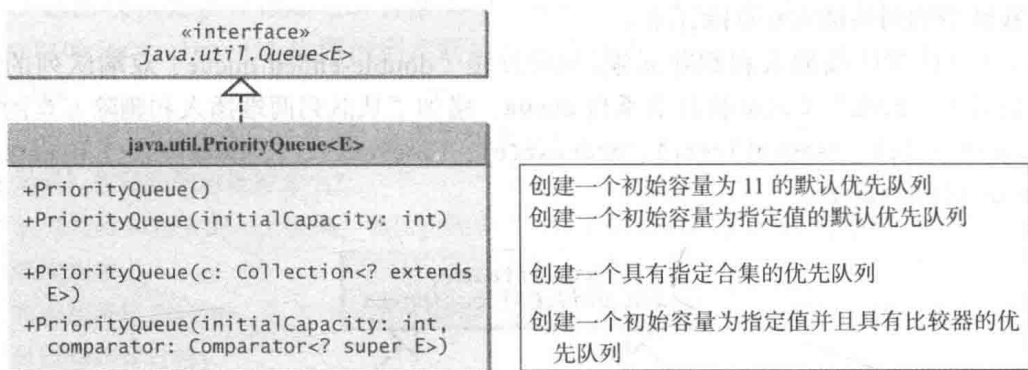


图 20-14 PriorityQueue 类实现了一个优先队列

程序清单 20-8 PriorityQueueDemo.java

```

1  import java.util.*;
2
3  public class PriorityQueueDemo {
4      public static void main(String[] args) {
5          PriorityQueue<String> queue1 = new PriorityQueue<>();
6          queue1.offer("Oklahoma");
7          queue1.offer("Indiana");
8          queue1.offer("Georgia");
9          queue1.offer("Texas");
10
11         System.out.println("Priority queue using Comparable:");
12         while (queue1.size() > 0) {
13             System.out.print(queue1.remove() + " ");
14         }
15
16         PriorityQueue<String> queue2 = new PriorityQueue(
17             4, Collections.reverseOrder());
18         queue2.offer("Oklahoma");
19         queue2.offer("Indiana");
20         queue2.offer("Georgia");
21         queue2.offer("Texas");
22
23         System.out.println("\nPriority queue using Comparator:");
24         while (queue2.size() > 0) {
25             System.out.print(queue2.remove() + " ");
26         }
27     }
28 }
  
```

```

Priority queue using Comparable:
Georgia Indiana Oklahoma Texas
Priority queue using Comparator:
Texas Oklahoma Indiana Georgia
  
```

✓ 复习题

- 20.29 java.util.Queue 是 java.util.Collection、java.util.Set 或 java.util.List 的子接口吗？LinkedList 实现了 Queue 吗？
- 20.30 如何创建一个整数优先队列？默认情况下，元素如何以优先队列排序？在优先队列中，拥有最小数值的元素被赋予最高优先级吗？
- 20.31 如何创建一个将元素的自然顺序颠倒的优先队列？

20.10 示例学习：表达式求值

要点提示：栈可以用于进行表达式求值。

栈和队列具有许多应用。本节给出一个使用栈来对表达式求值的应用。你可以从 Google 输入一个算术表达式来求值，如图 20-15 所示。

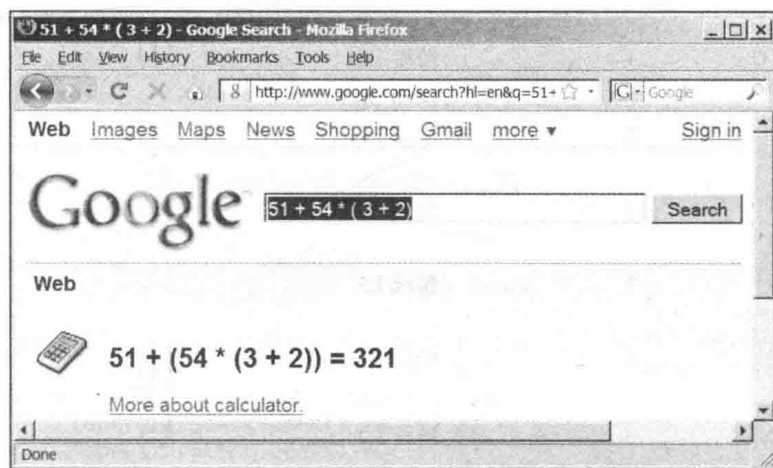


图 20-15 可以使用 Google 搜索引擎来对算术表达式求值

Google 是如何求值的呢？本节给出一个程序，对具有多个操作符和括号的复合表达式（compound expression）求值（例如， $(15 + 2) * 34 - 2$ ）。简化起见，假设操作数都是整数，并且操作符是 +、-、*、/ 四种之一。

这个问题可以使用两个栈来解决，命名为 `operandStack` 和 `operatorStack`，分别用于存储操作数和操作符。操作数和操作符在被处理前被压入栈中。当一个操作符被处理时，它从 `operatorStack` 中弹出，并应用于 `operandStack` 的前面两个操作数（两个操作数是从 `operandStack` 中弹出的）。结果数值被压回 `operandStack`。

这个算法分两个阶段进行：

阶段 1：扫描表达式

程序从左到右扫描表达式，提取出操作数、操作符以及括号。

- 1.1 如果提取的项是操作数，则将其压入 `operandStack`。
- 1.2 如果提取的项是 + 或 - 运算符，处理在 `operatorStack` 栈顶的所有运算符，将提取出的运算符压入 `operatorStack`。
- 1.3 如果提取的项目是 * 或 / 运算符，处理在 `operatorStack` 栈顶的所有 * 和 / 运算符，将提取出的运算符压入 `operatorStack`。
- 1.4 如果提取的项是 “(” 符号，将它压入 `operatorStack`。
- 1.5 如果提取的项是 “)” 符号，重复处理来自 `operatorStack` 栈顶的运算符，直到看到栈上的 “(” 符号。

阶段 2：清除栈

重复处理来自 `operatorStack` 栈顶的运算符，直到 `operatorStack` 为空为止。

表 20-1 显示了如何应用该算法来计算表达式 $(1+2)*4-3$ 。

表 20-1 对一个表达式求值

表达式	扫描	动作	operandStack	operatorStack
(1 + 2) * 4 - 3 ↑	(阶段 1.4	<div></div>	<div>(</div>
(1 + 2) * 4 - 3 ↑	1	阶段 1.1	<div>1</div>	<div>(</div>
(1 + 2) * 4 - 3 ↑	+	阶段 1.2	<div>1</div>	<div>+</div>
(1 + 2) * 4 - 3 ↑	2	阶段 1.1	<div>2</div> <div>1</div>	<div>(</div>
(1 + 2) * 4 - 3 ↑)	阶段 1.5	<div>3</div>	<div></div>
(1 + 2) * 4 - 3 ↑	*	阶段 1.3	<div>3</div>	<div>*</div>
(1 + 2) * 4 - 3 ↑	4	阶段 1.1	<div>4</div> <div>3</div>	<div>*</div>
(1 + 2) * 4 - 3 ↑	-	阶段 1.2	<div>12</div>	<div>-</div>
(1 + 2) * 4 - 3 ↑	3	阶段 1.1	<div>3</div> <div>12</div>	<div>-</div>
(1 + 2) * 4 - 3 ↑	无	阶段 2	<div>9</div>	<div></div>

程序清单 20-9 给出这个程序。图 20-16 给出一些样本输出。

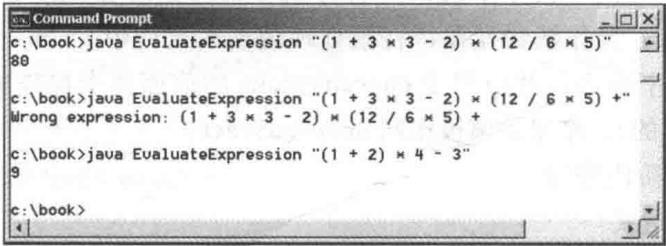


图 20-16 程序将一个表达式作为命令行参数

程序清单 20-9 EvaluateExpression.java

```
1 import java.util.Stack;
2
3 public class EvaluateExpression {
4     public static void main(String[] args) {
5         // Check number of arguments passed
6         if (args.length != 1) {
7             System.out.println(
8                 "Usage: java EvaluateExpression \"expression\"");
9             System.exit(1);
10        }
11
12        try {
13            System.out.println(evaluateExpression(args[0]));
14        }
15        catch (Exception ex) {
16            System.out.println("Wrong expression: " + args[0]);
17        }
18    }
19 }
```

```

18 }
19
20 /** Evaluate an expression */
21 public static int evaluateExpression(String expression) {
22     // Create operandStack to store operands
23     Stack<Integer> operandStack = new Stack<>();
24
25     // Create operatorStack to store operators
26     Stack<Character> operatorStack = new Stack<>();
27
28     // Insert blanks around (, ), +, -, /, and *
29     expression = insertBlanks(expression);
30
31     // Extract operands and operators
32     String[] tokens = expression.split(" ");
33
34     // Phase 1: Scan tokens
35     for (String token: tokens) {
36         if (token.length() == 0) // Blank space
37             continue; // Back to the while loop to extract the next token
38         else if (token.charAt(0) == '+' || token.charAt(0) == '-') {
39             // Process all +, -, *, / in the top of the operator stack
40             while (!operatorStack.isEmpty() &&
41                 (operatorStack.peek() == '+' ||
42                  operatorStack.peek() == '-' ||
43                  operatorStack.peek() == '*' ||
44                  operatorStack.peek() == '/')) {
45                 processAnOperator(operandStack, operatorStack);
46             }
47
48             // Push the + or - operator into the operator stack
49             operatorStack.push(token.charAt(0));
50         }
51         else if (token.charAt(0) == '*' || token.charAt(0) == '/') {
52             // Process all *, / in the top of the operator stack
53             while (!operatorStack.isEmpty() &&
54                 (operatorStack.peek() == '*' ||
55                  operatorStack.peek() == '/')) {
56                 processAnOperator(operandStack, operatorStack);
57             }
58
59             // Push the * or / operator into the operator stack
60             operatorStack.push(token.charAt(0));
61         }
62         else if (token.trim().charAt(0) == '(') {
63             operatorStack.push('('); // Push '(' to stack
64         }
65         else if (token.trim().charAt(0) == ')') {
66             // Process all the operators in the stack until seeing '('
67             while (operatorStack.peek() != '(') {
68                 processAnOperator(operandStack, operatorStack);
69             }
70
71             operatorStack.pop(); // Pop the '(' symbol from the stack
72         }
73         else { // An operand scanned
74             // Push an operand to the stack
75             operandStack.push(new Integer(token));
76         }
77     }
78
79     // Phase 2: Process all the remaining operators in the stack
80     while (!operatorStack.isEmpty()) {

```

```

81     processAnOperator(operandStack, operatorStack);
82 }
83
84 // Return the result
85 return operandStack.pop();
86 }
87
88 /** Process one operator: Take an operator from operatorStack and
89  * apply it on the operands in the operandStack */
90 public static void processAnOperator(
91     Stack<Integer> operandStack, Stack<Character> operatorStack) {
92     char op = operatorStack.pop();
93     int op1 = operandStack.pop();
94     int op2 = operandStack.pop();
95     if (op == '+')
96         operandStack.push(op2 + op1);
97     else if (op == '-')
98         operandStack.push(op2 - op1);
99     else if (op == '*')
100         operandStack.push(op2 * op1);
101     else if (op == '/')
102         operandStack.push(op2 / op1);
103 }
104
105 public static String insertBlanks(String s) {
106     String result = "";
107
108     for (int i = 0; i < s.length(); i++) {
109         if (s.charAt(i) == '(' || s.charAt(i) == ')' ||
110             s.charAt(i) == '+' || s.charAt(i) == '-' ||
111             s.charAt(i) == '*' || s.charAt(i) == '/')
112             result += " " + s.charAt(i) + " ";
113         else
114             result += s.charAt(i);
115     }
116
117     return result;
118 }
119 }

```

可以使用本书提供的 `GenericStack` 类或者定义在 Java API 中的 `java.util.Stack` 类来创建栈。本示例使用 `java.util.Stack` 类。如果替换成 `GenericStack`，程序依然可以运行。

该程序将一个表达式以一个字符串的形式作为命令行参数。

`evaluateExpression` 方法创建两个栈 `operandStack` 和 `operatorStack` (第 23 和 26 行)，并且提取被空格分隔的操作数、操作符以及括号 (第 29 ~ 32 行)。`insertBlanks` 方法用于保证操作数、操作符以及括号被至少一个空格分隔 (第 29 行)。

程序在 `for` 循环中扫描每个标记 (第 35 ~ 77 行)。如果标记是空的，那就跳过它 (第 37 行)。如果标记是一个操作数，那就将它压入 `operandStack` (第 75 行)。如果标记是一个 `+` 或 `-` 运算符 (第 38 行)，就处理在 `operatorStack` 栈顶的所有运算符 (如果有) (第 40 ~ 46 行)，并将新扫描到的运算符压入栈中 (第 49 行)。如果标记是一个 `*` 或 `/` 运算符 (第 51 行)，就处理在 `operatorStack` 栈顶的所有 `*` 和 `/` 运算符 (如果有) (第 53 ~ 57 行)，并将新扫描到的运算符压入栈中 (第 60 行)。如果标记是一个 “(” 符号 (第 62 行)，将它压入 `operatorStack`。如果标记是一个 “)” 符号 (第 65 行)，处理来自 `operatorStack` 栈顶的所有运算符，直到看到 “)” 符号 (第 67 ~ 69 行) 为止，然后从栈中弹出 “)” 符号。

在考虑完所有的标记之后，程序处理 `operatorStack` 中剩余的运算符 (第 80 ~ 82 行)。

`processAnOperator` 方法 (第 90 ~ 103 行) 用来处理一个运算符。该方法从 `operatorStack` 中弹出一个运算符 (第 92 行) 并且从 `operandStack` 中弹出两个操作数 (第 93 ~ 94 行)。依据所弹出的运算符, 该方法完成对应的操作, 然后将操作结果压回 `operandStack` 中 (第 96、98、100 和 102 行)。

✓ 复习题

- 20.23 `EvaluateExpression` 程序可以对表达式 `"1 + 2"`、`"1 + 2"`、`"(1) + 2"`、`"((1)) + 2"` 以及 `"(1 + 2)"` 求值吗?
- 20.24 使用 `EvaluateExpression` 程序对 `"3 + (4 + 5)*(3 + 5) + 4 * 5"` 求值时, 给出栈中内容的变化。
- 20.25 如果输入表达式 `"4 + 5 5 5"`, 程序将显示 10。如果修改这个问题?

关键术语

collection (合集)

linked list (链表)

comparator (比较器)

list (线性表)

convenience abstract class (便利抽象类)

priority queue (优先队列)

data structure (数据结构)

queue (队列)

本章小结

1. Java 合集框架支持集合、线性表、队列和映射表, 它们分别定义在接口 `Set`、`List`、`Queue` 和 `Map` 中。
2. 线性表用于存储一个有序的元素合集。
3. 除去 `PriorityQueue`, Java 合集框架中的所有实例类都实现了 `Cloneable` 和 `Serializable` 接口。所以, 它们的实例都是可克隆和可序列化的。
4. 若要在合集中存储重复的元素, 就需要使用线性表。线性表不仅可以存储重复的元素, 而且允许用户指定存储的位置。用户可以通过下标来访问线性表中的元素。
5. Java 合集框架支持两种类型的线性表: 数组线性表 `ArrayList` 和链表 `LinkedList`。`ArrayList` 是实现 `List` 接口的可变大小的数组。`ArrayList` 中的所有方法都是在 `List` 接口中定义的。`LinkedList` 是实现 `List` 接口的一个链表。除了实现了 `List` 接口, 该类还提供了可从线性表两端提取、插入以及删除元素的方法。
6. `Comparator` 可以用于比较没有实现 `Comparable` 接口的类的对象。
7. `Vector` 类继承了 `AbstractList` 类。从 Java 2 开始, `Vector` 类和 `ArrayList` 是一样的, 所不同的是它所包含的访问和修改向量的方法是同步的。`Stack` 类继承了 `Vector` 类, 并且提供了几种对栈进行操作的方法。
8. `Queue` 接口表示队列。`PriorityQueue` 类为优先队列实现 `Queue` 接口。

测试题

回答位于网址 www.cs.armstrong.edu/liang/intro10e/quiz.html 的本章测试题。

编程练习题

20.2 ~ 20.7 节

- *20.1 (按字母序的升序显示单词) 编写一个程序, 从文本文件读取单词, 并按字母的升序显示所有的

单词（可以重复）。单词必须以字母开始。文本文件作为命令行参数传递。

- *20.2（对链表中的数字进行排序）编写一个程序，让用户从图形用户界面输入数字，然后在文本区域显示它们，如图 20-17a 所示。使用链表存储这些数字，但不要存储重复的数值。添加按钮 Sort、Shuffle 和 Reverse，分别对这个线性表进行排序、打乱顺序与颠倒顺序操作。

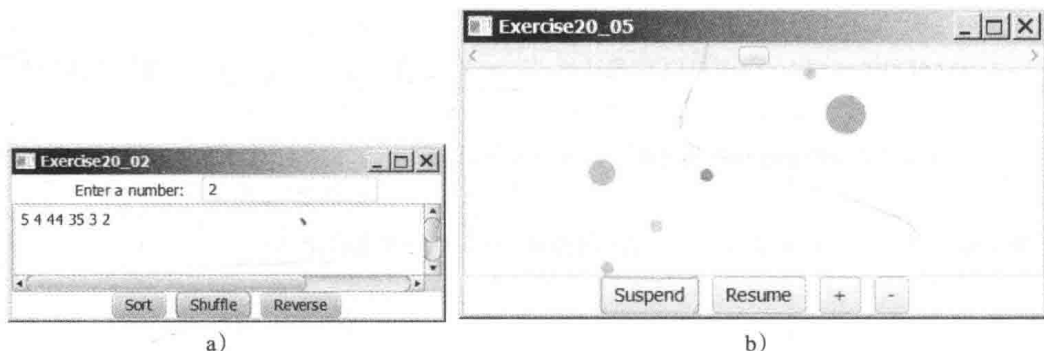


图 20-17 a) 数字保存在线性表中并显示在一个文本区域内；b) 相撞的球结合在一起

- *20.3（猜首府）改写编程练习题 8.37，保存州和首府的匹配对，以随机显示问题。
- *20.4（对面板上的点进行排序）编写一个程序，满足下面的要求：
- 定义一个名为 `Point` 的类，它的两个数据域 `x` 和 `y` 分别表示点的 `x` 坐标和 `y` 坐标。实现 `Comparable` 接口用于比较点的 `x` 坐标。如果两个点的 `x` 坐标一样，则比较它们的 `y` 坐标。
 - 定义一个名为 `CompareY` 的类实现 `Comparator<Point>`。实现 `compare` 方法来通过 `y` 坐标值比较两个点。如果 `y` 坐标值一样，则比较它们的 `x` 坐标值。
 - 随机创建 100 个点，然后使用 `Arrays.sort` 方法分别以它们 `x` 坐标的升序和 `y` 坐标的升序显示这些点。
- ***20.5（合并碰撞的弹球）20.7 节的示例中显示了多个弹球。扩充该例子来进行碰撞检测。一旦两个球相撞，移除后面加入面板的那个球，并且将它的半径加到另外一个球上，如图 20-17b 所示。使用 `Suspend` 按钮来暂停动画，以及 `Resume` 按钮来继续动画。添加一个鼠标按下处理器，从而在鼠标按在球上的时候移除这个球。
- 20.6（在链表上使用遍历器）编写一个测试程序，在一个链表上存储 500 万个整数，测试分别使用 `iterator` 和使用 `get(index)` 方法的遍历时间。
- ***20.7（游戏：猜字游戏）编程练习题 7.35 给出了流行的猜字游戏的控制台版本。编写一个 GUI 程序让用户来玩这个游戏。用户通过一次输入一个字母来猜单词，如图 20-18 所示。如果用户 7 次都没猜对，被吊的人就摆动起来。一旦完成一个单词，用户就可以按 `Enter` 键继续猜另一个单词。
- **20.8（游戏：彩票）修改编程练习题 3.15，如果用户输入的两个数字在彩票号码之中，增加额外的 2000 美元。（提示：对彩票中的三个数字和用户输入的三个数字进行排序，并分别存入两个线性表，然后使用 `Collection` 的 `containsAll` 方法来检测用户输入的两个数字是否在彩票数字中。）
- 20.8 ~ 20.10 节
- ***20.9（首先移除最大的球）修改程序清单 20-6，使得一个球在被创建的时候赋给一个 2 ~ 20 的随机半径。当单击“-”按钮时，最大的一个球被移除。
- 20.10（在优先队列上进行集合操作）创建两个优先队列，{"George", "Jim", "John", "Blake", "Kevin", "Michael"} 和 {"George", "Katie", "Kevin", "Michelle", "Ryan"}，求它们的并集、差集和交集。

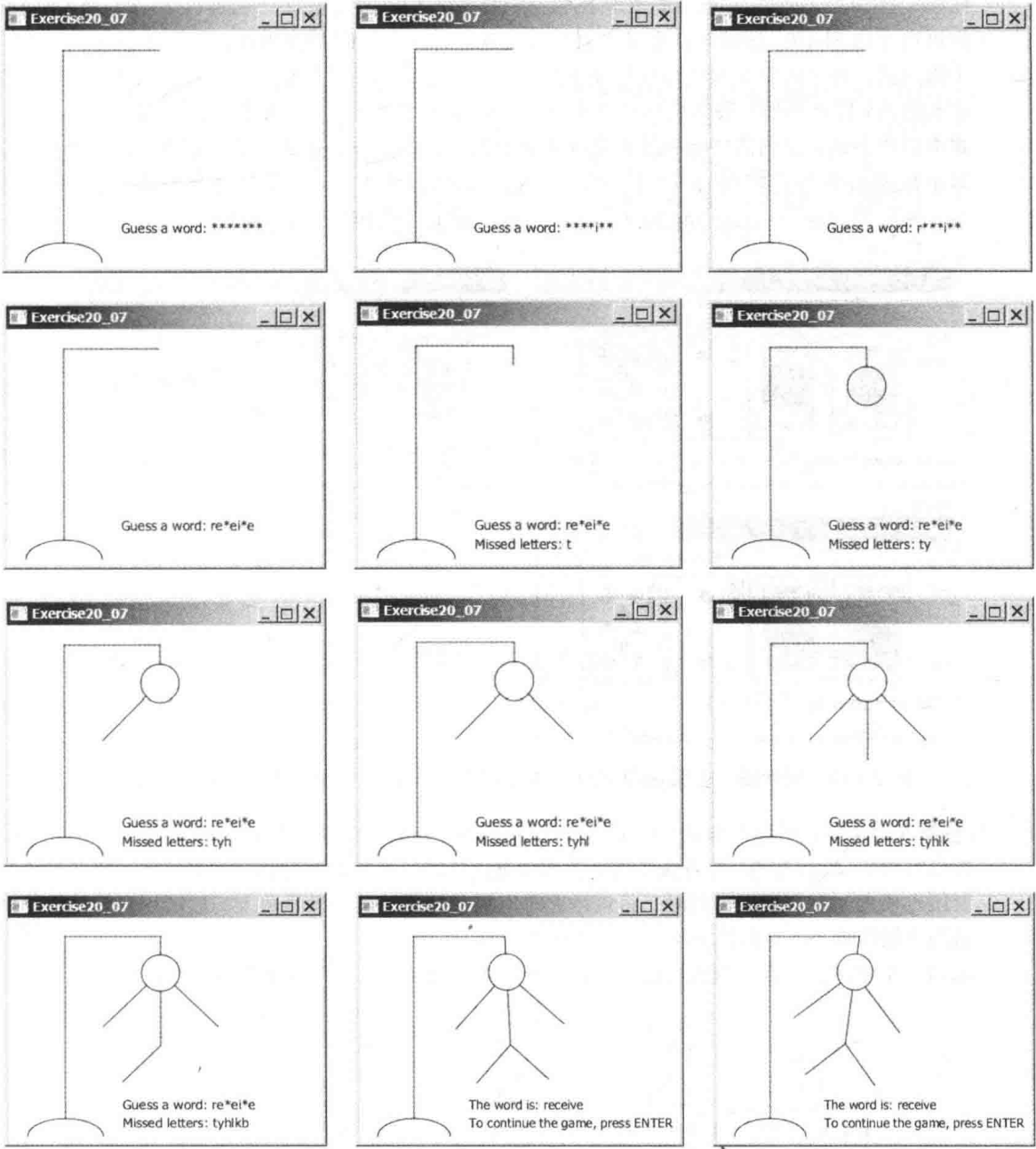


图 20-18 程序显示一个猜字游戏

*20.11 (编组符号匹配) Java 程序包含各种编组符号对，例如：

- 圆括号：(和)
- 花括号：{ 和 }
- 方括号：[和]

请注意编组符号不能交错。例如，(a{b})是不合法的。编写一个程序来检测一个 Java 源程序中是否编组符号都是正确匹配的。将源代码文件名字作为命令行参数传递。

20.12 (克隆 PriorityQueue) 定义 MyPriorityQueue 类，继承自 PriorityQueue 并实现 Cloneable 接口和实现 clone() 方法来克隆一个优先队列。

**20.13 (游戏：24 点扑克牌游戏) 24 点游戏是指从 52 张牌中任意选取 4 张扑克牌，如图 20-19 所示。

注意，将两个王排除在外。每张牌表示一个数字。A、K、Q 和 J 分别表示 1、13、12 和 11。你可以单击 Shuffle 按钮来获取 4 张新的扑克牌。输入这 4 张扑克牌牌面的 4 个数字构成的一个表达式。每个数字必须使用且只能使用一次。可以在表达式中使用运算符（加法、减法、乘法和除法）以及括号。表达式必须计算出 24。在输入表达式之后，单击 Verify 按钮来检查表达式中的数字是否是当前所选择的扑克牌牌面上的数，并检查表达式的结果是否正确。检查结果显示在 Shuffle 按钮前面的一个标签中。假设图像以黑桃、红心、方块和梅花的顺序存储在名为 1.png, 2.png, ..., 52.png 的文件中，这样，前 13 个图像就是黑桃的 1, 2, 3, ..., 13。

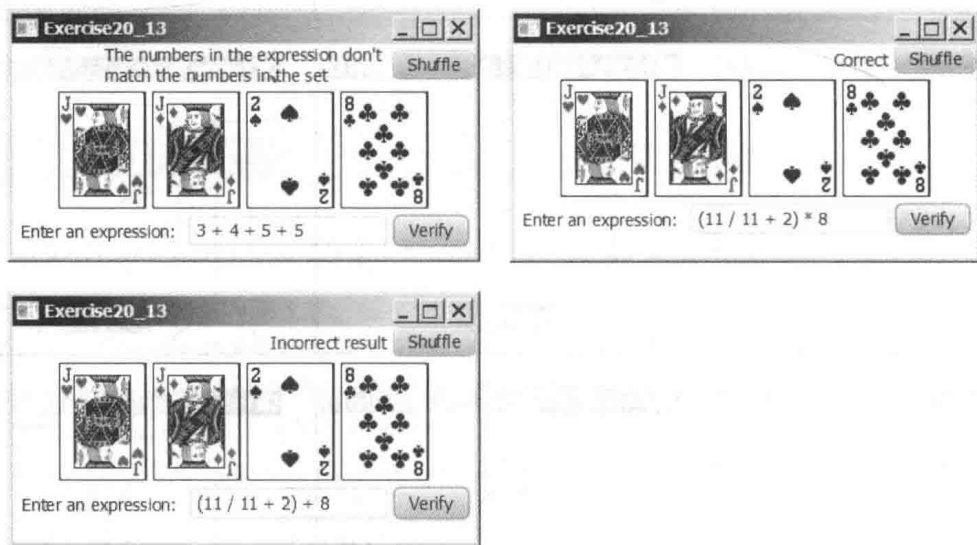
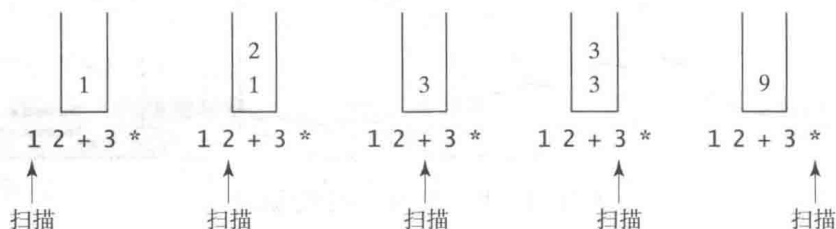


图 20-19 用户输入由牌面数字组成的表达式，并单击 Verify 按钮来检查结果

****20.14 (后缀表示法)** 后缀表示法是一种不使用括号编写表达式的方法。例如，表达式 $(1 + 2) * 3$ 可以写为 $1\ 2\ +\ 3\ *$ 。后缀表达式是使用栈来计算的。从左到右扫描后缀表达式，将变量或常量压入栈内，当遇到运算符时，将该运算符应用在栈顶的两个操作数上，然后用运算结果替换这两个操作数。下面的图演示了如何计算 $1\ 2\ +\ 3\ *$ 。

编写一个程序，计算后缀表达式，将后缀表达式作为一个字符串的命令行参数传递。



*****20.15 (游戏：24 点扑克牌游戏)** 改进编程练习题 20.13，如果表达式存在，那就让计算机显示它，如图 20-20 所示；否则，报告这样的表达式不存在。将显示验证结果的标签置于 UI 的底部。表达式必须使用所有 4 张扑克牌并且值等于 24。

****20.16 (将中缀转换为后缀)** 使用下面的方法头编写方法，将中缀表达式转换为一个后缀表达式：

```
public static String infixToPostfix(String expression)
```

例如，该方法可以将中缀表达式 $(1+2)*3$ 转换为 $1\ 2\ +\ 3\ *$ ，将 $2*(1+3)$ 转换为 $2\ 1\ 3\ +\ *$ 。

*****20.17 (游戏：24 点扑克牌游戏)** 此练习题是编程练习题 20.13 中描述的 24 点扑克牌游戏的变体。编写一个程序，检查是否有这 4 个给定数的 24 点的解决方案。该程序让用户输入 1 ~ 13 的 4 个

值，如图 20-21 所示。然后用户可以单击 Solve 按钮来显示解决方案，若不存在解决方案，就提示“不存在解决方案”。

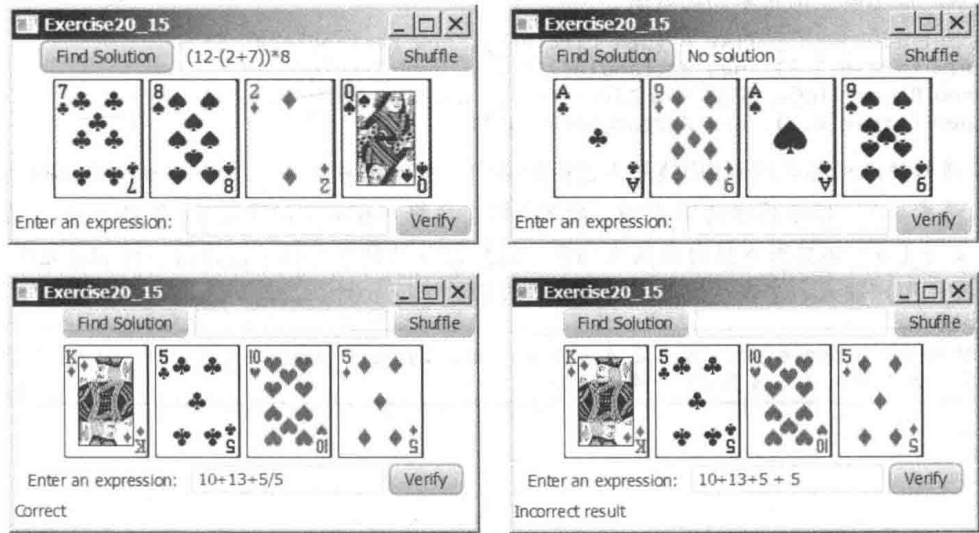


图 20-20 如果存在一个解决方案，程序可以自动找到它



图 20-21 用户输入 4 个数字，然后程序找出解决方案

*20.18（目录大小）程序清单 20-7 使用递归方法来找到一个目录大小。重写该方法，不使用递归。程序应该使用一个队列来存储一个目录下的所有子目录。算法可以如下描述：

```
long getSize(File directory) {
    long size = 0;
    add directory to the queue;

    while (queue is not empty) {
        Remove an item from the queue into t;
        if (t is a file)
            size += t.length();
        else
            add all the files and subdirectories under t into the
            queue;
    }

    return size;
}
```

***20.19（游戏：24 点游戏有解的比例）回顾编程练习题 20.13 介绍的 24 点游戏，从 52 张牌中选择 4 张牌，这 4 张牌可能没有能得到 24 点的解决方案。从 52 张牌中选择 4 张牌的所有可能的挑选次数是多少？在这些所有可能的挑选中，有多少可以得到 24 点？成功的几率（即可得到 24 点的挑选次数）/（所有可能的挑选次数）是多少？编写一个程序，找出这些答案。

*20.20（目录大小）重写编程练习题 18.28，使用栈而不是使用队列来解决这个问题。

*20.21（使用 Comparator）使用选择排序和比较器，编写以下通用的方法。

```
public static <E> void selectionSort(E[] list,
    Comparator<? super E> comparator)
```

编写一个测试程序，创建一个具有 10 个 `GeometricObject` 对象的数组，并且使用程序清单 20-4 介绍的 `GeometricObjectComparator` 调用该方法对元素进行排序。显示排好序的元素。使用以下语句来创建数组。

```
GeometricObject[] list = {new Circle(5), new Rectangle(4, 5),  
    new Circle(5.5), new Rectangle(2.4, 5), new Circle(0.5),  
    new Rectangle(4, 65), new Circle(4.5), new Rectangle(4.4, 1),  
    new Circle(6.5), new Rectangle(4, 5)};
```

*20.22 (非递归的汉诺塔实现) 使用栈而不是使用递归，实现程序清单 18-8 中的 `moveDisks` 方法。

**20.23 (表达式求值) 修改程序清单 20-9，增加指数运算符 \wedge 和求模运算符 $\%$ 。例如， $3 \wedge 2$ 等于 9， $3 \% 2$ 等于 1。运算符 \wedge 具有最高优先级，运算符 $\%$ 具有与 $*$ 和 $/$ 运算符一样的优先级。程序应该提示用户输入一个表达式。下面是一个程序的运行示例：


```
Enter an expression: (5 * 2 ^ 3 + 2 * 3 % 2) * 4 ↵ Enter  
(5 * 2 ^ 3 + 2 * 3 % 2) * 4 = 160
```

集合和映射表

教学目标

- 使用集合存储无序的、没有重复的元素（21.2 节）。
- 探究如何使用以及何时使用 `HashSet`（21.2.1 节）、`LinkedHashSet`（21.2.2 节）或者 `TreeSet`（21.2.3 节）来存储元素。
- 比较集合和线性表的性能（21.3 节）。
- 使用集合开发一个计算 Java 源文件中关键字数目的程序（21.4 节）。
- 区分 `Collection` 与 `Map`，并描述何时及如何使用 `HashMap`、`LinkedHashMap` 或者 `TreeMap` 来存储带键值的值（21.5 节）。
- 使用映射表开发一个计算文本文件中单词出现次数的程序（21.6 节）。
- 使用 `Collections` 类中的静态方法来获得单元素的集合、线性表和映射表，以及不可变的集合、线性表和映射表（21.7 节）。

21.1 引言


 **要点提示：**集合（set）是一个用于存储和处理无重复元素的高效数据结构。映射表（map）类似于目录，提供了使用键值快速查询和获取值的功能。

禁飞名单是一个由美国政府恐怖分子筛选检查中心创建和维护的一张表，列出了不允许搭乘商业飞机进出美国的人员名单。假设我们需要写一个程序，检验一个人是否在禁飞名单上，可以使用一个线性表来存储禁飞名单上面的名字。然而，用来实现这个程序的更有效的数据结构是集合（set）。

假设你的程序还需要存储禁飞名单上恐怖分子的详细信息，可以使用名字作为键值来获取诸如性别、身高、体重以及国籍等详细信息。映射表（map）是实现这种任务的有效数据结构。

本章介绍 Java 合集框架中的集合和映射表。

21.2 集合

 **要点提示：**可以使用集合的三个具体类 `HashSet`、`LinkedHashSet` 和 `TreeSet` 来创建集合。

`Set` 接口扩展了 `Collection` 接口，如图 20-1 所示。它没有引入新的方法或常量，只是规定 `Set` 的实例不包含重复的元素。实现 `Set` 的具体类必须确保不能向这个集合添加重复的元素。也就是说，在一个集合中，不存在元素 `e1` 和 `e2`，使得 `e1.equals(e2)` 的返回值为 `true`。

`AbstractSet` 类继承 `AbstractCollection` 类并部分实现 `Set` 接口。`AbstractSet` 类提供 `equals` 方法和 `hashCode` 方法的具体实现。一个集合的散列码是这个集合中所有元素散列码的和。由于 `AbstractSet` 类没有实现 `size` 方法和 `iterator` 方法，所以 `AbstractSet` 类是一

个抽象类。

Set 接口的三个具体类是：散列类 HashSet、链式散列集 LinkedHashMap 和树形集 TreeSet，如图 21-1 所示。

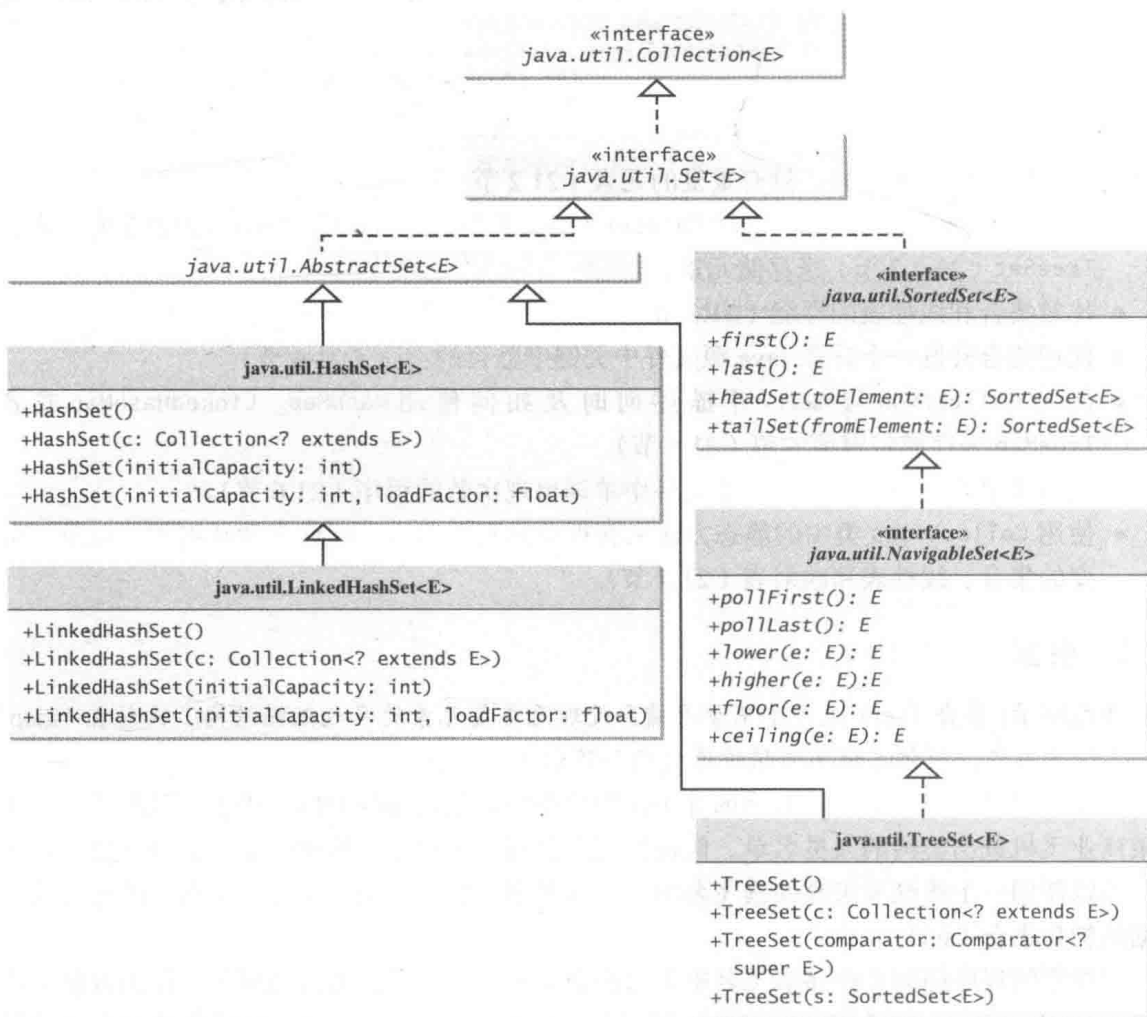


图 21-1 Java 合集框架提供三个具体集合类

21.2.1 HashSet

HashSet 类是一个实现了 Set 接口的具体类，可以使用它的无参构造方法来创建空的散列集（hash set），也可以由一个现有的合集创建散列集。默认情况下，初始容量为 16 而负载系数是 0.75。如果知道集合的大小，就可以在构造方法中指定初始容量和负载系数。否则，就使用默认的设置，负载系数的值在 0.0 ~ 1.0 之间。

在增加集合的容量之前，负载系数（load factor）测量该集合允许多满。当元素个数超过了容量与负载系数的乘积，容量就会自动翻倍。例如，如果容量是 16 而负载系数是 0.75，那么当尺寸达到 12（ $16 \times 0.75 = 12$ ）时，容量将会翻倍到 32。比较高的负载系数会降低空间开销，但是会增加查找时间。通常情况下，默认的负载系数是 0.75，它是在时间开销和空间开销上一个很好的权衡。我们将在第 27 章更深入地讨论负载系数。

HashSet 类可以用来存储互不相同的任何元素。考虑到效率的因素，添加到散列集中的对象必须以一种正确分散散列码的方式来实现 hashCode 方法。回顾在 Object 类中定义的 hashCode，如果两个对象相等，那么这两个对象的散列码必须一样。两个不相等的对象可能会有相同的散列码，因此你应该实现 hashCode 方法以避免出现太多这样的情况。Java API 中的大多数类都实现了 hashCode 方法。例如，Integer 类中的 hashCode 方法返回它的 int 值，Character 类中的 hashCode 方法返回这个字符的统一码，String 类中的 hashCode 方法返回 $s_0 * 31^{(n-1)} + s_1 * 31^{(n-2)} + \dots + s_{n-1}$ ，其中 s_i 是 $s.charAt(i)$ 。

程序清单 21-1 给出的程序创建了一个散列集来存储字符串，并且使用一个 foreach 循环来遍历这个集合中的元素。

程序清单 21-1 TestHashSet.java

```

1 import java.util.*;
2
3 public class TestHashSet {
4     public static void main(String[] args) {
5         // Create a hash set
6         Set<String> set = new HashSet<>();
7
8         // Add strings to the set
9         set.add("London");
10        set.add("Paris");
11        set.add("New York");
12        set.add("San Francisco");
13        set.add("Beijing");
14        set.add("New York");
15
16        System.out.println(set);
17
18        // Display the elements in the hash set
19        for (String s: set) {
20            System.out.print(s.toUpperCase() + " ");
21        }
22    }
23 }
```

[San Francisco, New York, Paris, Beijing, London]
 SAN FRANCISCO NEW YORK PARIS BEIJING LONDON

该程序将多个字符串添加到集合中（第 9 ~ 14 行）。New York 被添加多次，但是只有一个被存储，因为集合不允许有重复的元素。

如输出所示，字符串没有按照它们被插入集合时的顺序存储，因为散列集中的元素是没有特定的顺序的。要强加给它们一个顺序，就需要使用 LinkedHashMap 类，这个类将在下一节中介绍。

回顾前面提到的，Collection 接口继承 Iterable 接口，因此集合中的元素是可遍历的。使用了 foreach 循环来遍历集合中的所有元素（第 19 ~ 21 行）。

由于一个集合是 Collection 的一个实例，因此，所有定义在 Collection 中的方法都可以用在集合上。程序清单 21-2 给出一个应用 Collection 接口中方法的例子。

程序清单 21-2 TestMethodsInCollection.java

```

1 public class TestMethodsInCollection {
2     public static void main(String[] args) {
3         // Create set1
```



```
4 java.util.Set<String> set1 = new java.util.HashSet<>();
5
6 // Add strings to set1
7 set1.add("London");
8 set1.add("Paris");
9 set1.add("New York");
10 set1.add("San Francisco");
11 set1.add("Beijing");
12
13 System.out.println("set1 is " + set1);
14 System.out.println(set1.size() + " elements in set1");
15
16 // Delete a string from set1
17 set1.remove("London");
18 System.out.println("\nset1 is " + set1);
19 System.out.println(set1.size() + " elements in set1");
20
21 // Create set2
22 java.util.Set<String> set2 = new java.util.HashSet<>();
23
24 // Add strings to set2
25 set2.add("London");
26 set2.add("Shanghai");
27 set2.add("Paris");
28 System.out.println("\nset2 is " + set2);
29 System.out.println(set2.size() + " elements in set2");
30
31 System.out.println("\nIs Taipei in set2? "
32 + set2.contains("Taipei"));
33
34 set1.addAll(set2);
35 System.out.println("\nAfter adding set2 to set1, set1 is "
36 + set1);
37
38 set1.removeAll(set2);
39 System.out.println("After removing set2 from set1, set1 is "
40 + set1);
41
42 set1.retainAll(set2);
43 System.out.println("After removing non-common elements in set2 "
44 + "from set1, set1 is " + set1);
45 }
46 }
```

```
set1 is [San Francisco, New York, Paris, Beijing, London]
5 elements in set1

set1 is [San Francisco, New York, Paris, Beijing]
4 elements in set1

set2 is [Shanghai, Paris, London]
3 elements in set2

Is Taipei in set2? false

After adding set2 to set1, set1 is
[San Francisco, New York, Shanghai, Paris, Beijing, London]

After removing set2 from set1, set1 is
[San Francisco, New York, Beijing]

After removing non-common elements in set2 from set1, set1 is []
```

该程序创建了两个集合（第 4 和 22 行）。方法 `size()` 返回一个集合中的元素个数（第

14 行)。第 17 行

```
set1.remove("London");
```

从 set1 中删除 London。

方法 contains (第 32 行) 检测一个元素是否在某个集合中。

第 34 行

```
set1.addAll(set2);
```

将 set2 添加给 set1。这样, set1 就变成 [San Francisco, New York, Shanghai, Paris, Beijing, London]。

第 38 行

```
set1.removeAll(set2);
```

从 set1 中删除 set2。这样, set1 就变成 [San Francisco, New York, Beijing]。

第 42 行

```
set1.retainAll(set2);
```

保留和 set1 共有的元素。因为 set1 和 set2 没有公共的元素, 所以 set1 就变成空的。

21.2.2 LinkedHashSet

LinkedHashSet 用一个链表实现来扩展 HashSet 类, 它支持对集合内的元素排序。HashSet 中的元素是没有被排序的, 而 LinkedHashSet 中的元素可以按照它们插入集合的顺序提取。LinkedHashSet 对象是可以使用它的 4 个构造方法之一来创建的, 如图 21-1 所示。这些构造方法类似于 HashSet 的构造方法。

程序清单 21-3 给出一个测试 LinkedHashSet 的程序。这个程序只是用 LinkedHashSet 来替换程序清单 21-1 中的 HashSet。

程序清单 21-3 TestLinkedHashSet.java

```
1 import java.util.*;
2
3 public class TestLinkedHashSet {
4     public static void main(String[] args) {
5         // Create a hash set
6         Set<String> set = new LinkedHashSet<>();
7
8         // Add strings to the set
9         set.add("London");
10        set.add("Paris");
11        set.add("New York");
12        set.add("San Francisco");
13        set.add("Beijing");
14        set.add("New York");
15
16        System.out.println(set);
17
18        // Display the elements in the hash set
19        for (String element: set)
20            System.out.print(element.toLowerCase() + " ");
21    }
22 }
```

```
[London, Paris, New York, San Francisco, Beijing]
london paris new york san francisco beijing
```

第 6 行创建了一个 `LinkedHashSet` 对象。如输出中所示，字符串按照它们插入集合的顺序存储。由于 `LinkedHashSet` 是一个集合，所以它不能存储重复的元素。

`LinkedHashSet` 保持了元素插入时的顺序。要强加一个不同的顺序（例如，升序或降序），可以使用下一节介绍的 `TreeSet` 类。

提示： 如果不需要维护元素被插入的顺序，就应该使用 `HashSet`，它会比 `LinkedHashSet` 更加高效。

21.2.3 TreeSet

`SortedSet` 是 `Set` 的一个子接口，它可以确保集合中的元素是有序的。另外，它还提供方法 `first()` 和 `last()` 以返回集合中的第一个元素和最后一个元素，以及方法 `headSet(toElement)` 和 `tailSet(fromElement)` 以分别返回集合中元素小于 `toElement` 和大于或等于 `fromElement` 的那一部分。

`NavigableSet` 扩展了 `SortedSet`，并提供导航方法 `lower(e)`、`floor(e)`、`ceiling(e)` 和 `higher(e)` 以分别返回小于、小于或等于、大于或等于以及大于一个给定元素的元素。如果没有这样的元素，方法就返回 `null`。方法 `pollFirst()` 和 `pollLast()` 会分别删除和返回树形集中的第一个元素和最后一个元素。

`TreeSet` 实现了 `SortedSet` 接口。为了创建 `TreeSet` 对象，可以使用如图 21-1 所示的构造方法。只要对象是可以互相比较的，就可以将它们添加到一个树形集（tree set）中。

如 20.5 节所讨论的，元素可以有两种方法进行比较：使用 `Comparable` 接口或者 `Comparator` 接口。

程序清单 21-4 给出使用 `Comparable` 接口对元素进行排序的例子。前面的程序清单 21-3 中的例子以字符串插入的顺序显示所有的字符串。这个例子重写前面的例子，使用 `TreeSet` 类按照字母顺序来显示这些字符串。

程序清单 21-4 TestTreeSet.java

```
1 import java.util.*;
2
3 public class TestTreeSet {
4     public static void main(String[] args) {
5         // Create a hash set
6         Set<String> set = new HashSet<>();
7
8         // Add strings to the set
9         set.add("London");
10        set.add("Paris");
11        set.add("New York");
12        set.add("San Francisco");
13        set.add("Beijing");
14        set.add("New York");
15
16        TreeSet<String> treeSet = new TreeSet<>(set);
17        System.out.println("Sorted tree set: " + treeSet);
18
19        // Use the methods in SortedSet interface
20        System.out.println("first(): " + treeSet.first());
21        System.out.println("last(): " + treeSet.last());
22        System.out.println("headSet(\"New York\"): " +
```

```
23     treeSet.headSet("New York"));
24     System.out.println("tailSet(\"New York\"): " +
25         treeSet.tailSet("New York"));
26
27     // Use the methods in NavigableSet interface
28     System.out.println("lower(\"P\"): " + treeSet.lower("P"));
29     System.out.println("higher(\"P\"): " + treeSet.higher("P"));
30     System.out.println("floor(\"P\"): " + treeSet.floor("P"));
31     System.out.println("ceiling(\"P\"): " + treeSet.ceiling("P"));
32     System.out.println("pollFirst(): " + treeSet.pollFirst());
33     System.out.println("pollLast(): " + treeSet.pollLast());
34     System.out.println("New tree set: " + treeSet);
35 }
36 }
```

```
Sorted tree set: [Beijing, London, New York, Paris, San Francisco]
first(): Beijing
last(): San Francisco
headSet("New York"): [Beijing, London]
tailSet("New York"): [New York, Paris, San Francisco]
lower("P"): New York
higher("P"): Paris
floor("P"): New York
ceiling("P"): Paris
pollFirst(): Beijing
pollLast(): San Francisco
New tree set: [London, New York, Paris]
```

本例创建了一个由字符串填充的散列集，然后创建一个由相同字符串构成的树形集，使用 `Comparable` 接口中的 `compareTo` 方法对树形集中的字符串进行排序。

当使用语句 `new TreeSet<String>(Set)` (第 16 行) 从一个 `HashSet` 对象创建一个 `TreeSet` 对象时，集合中的元素被排序。可以改写这个程序，使用 `TreeSet` 的无参构造方法来创建一个 `TreeSet` 的实例，然后将字符串添加到这个实例中。

`treeSet.first()` 返回 `treeSet` 中的第一个元素 (第 20 行)。`treeSet.last()` 返回 `treeSet` 中的最后一个元素 (第 21 行)。`treeSet.headSet("New York")` 返回 `treeSet` 中 `New York` 之前的那些元素 (第 22 ~ 23 行)。`treeSet.tailSet("New York")` 返回 `treeSet` 中 `New York` 及其后的元素 (第 24 ~ 25 行)。

`treeSet.lower("P")` 返回 `treeSet` 中小于 `P` 的最大元素 (第 28 行)。`treeSet.higher("P")` 返回 `treeSet` 中大于 `P` 的最小元素 (第 29 行)。`treeSet.floor("P")` 返回 `treeSet` 中小于或等于 `P` 的最大元素 (第 30 行)。`treeSet.ceiling("P")` 返回 `treeSet` 中大于或等于 `P` 的最小元素 (第 31 行)。`treeSet.pollFirst()` 删除 `treeSet` 中的第一个元素，并返回被删除的元素 (第 32 行)。`treeSet.pollLast()` 删除 `treeSet` 中的最后一个元素，并返回被删除的元素 (第 33 行)。

注意：Java 合集框架中的所有具体类 (参见图 20-1) 都至少有两个构造方法：一个是创建空合集的无参构造方法，另一个是用某个合集来创建实例的构造方法。这样，`TreeSet` 类中就含有从合集 `c` 创建 `TreeSet` 对象的构造方法 `TreeSet(Collection c)`。在这个例子中，`new TreeSet<>(set)` 方法从合集 `set` 创建了 `TreeSet` 的一个实例。

提示：当更新一个集合时，如果不需要保持元素的排序关系，就应该使用散列集，因为在散列集中插入和删除元素所花的时间比较少。当需要一个排好序的集合时，可以从这个散列集创建一个树形集。

如果使用无参构造方法创建一个 `TreeSet`，则会假定元素的类实现了 `Comparable` 接

口, 并使用 `compareTo` 方法来比较集合中的元素。要使用 `comparator`, 则必须用构造方法 `TreeSet(Comparator comparator)`, 使用比较器中的 `compare` 方法来创建一个排好序的集合。

程序清单 21-5 给出了一个程序, 演示了如何使用 `Comparator` 接口来对树形集中的元素进行排序。

程序清单 21-5 TestTreeSetWithComparator.java

```

1  import java.util.*;
2
3  public class TestTreeSetWithComparator {
4      public static void main(String[] args) {
5          // Create a tree set for geometric objects using a comparator
6          Set<GeometricObject> set =
7              new TreeSet<>(new GeometricObjectComparator());
8          set.add(new Rectangle(4, 5));
9          set.add(new Circle(40));
10         set.add(new Circle(40));
11         set.add(new Rectangle(4, 1));
12
13         // Display geometric objects in the tree set
14         System.out.println("A sorted set of geometric objects");
15         for (GeometricObject element: set)
16             System.out.println("area = " + element.getArea());
17     }
18 }

```

```

A sorted set of geometric objects
area = 4.0
area = 20.0
area = 5021.548245743669

```

`GeometricObjectComparator` 类在程序清单 20-4 中定义。程序创建了一个几何对象的树形集, 并使用 `GeometricObjectComparator` 来比较集合中的元素 (第 6 ~ 7 行)。

`Circle` 类和 `Rectangle` 类已经在 13.2 节中定义, 它们都是几何类 `GeometricObject` 的子类, 被加入到集合中 (第 8 ~ 11 行)。

两个半径相同的圆都被添加到树形集的集合内 (第 9 ~ 10 行), 但是只存储一个, 因为这两个圆是相等的, 而集合内不允许有重复的元素。

✓ 复习题

- 21.1 如何创建 `Set` 的一个实例? 如何在集合内插入一个新元素? 如何从集合中删除一个元素? 如何获取一个集合的大小?
- 21.2 如果两个对象 `o1` 和 `o2` 是相等的, 那么 `o1.equals(o2)` 和 `o1.hashCode() == o2.hashCode()` 分别为多少?
- 21.3 `HashSet`、`LinkedHashSet` 和 `TreeSet` 之间的区别是什么?
- 21.4 如何遍历集合中的元素?
- 21.5 如何使用 `Comparable` 接口中的方法 `compareTo` 对集合内的元素进行排序? 如何使用 `Comparator` 接口对集合内的元素进行排序? 如果向树形集内添加一个不能与已有元素进行比较的元素, 会发生什么情况?
- 21.6 假设 `set1` 是包含字符串 `red`、`yellow`、`green` 的集合, 而 `set2` 是包含字符串 `red`、`yellow`、`blue` 的集合, 回答下面的问题:
 - 执行完 `set1.addAll(set2)` 方法之后, 集合 `set1` 和 `set2` 分别变成了什么?
 - 执行完 `set1.add(set2)` 方法之后, 集合 `set1` 和 `set2` 分别变成了什么?

- 执行完 `set1.removeAll(set2)` 方法之后, 集合 `set1` 和 `set2` 分别变成了什么?
- 执行完 `set1.remove(set2)` 方法之后, 集合 `set1` 和 `set2` 分别变成了什么?
- 执行完 `set1.retainAll(set2)` 方法之后, 集合 `set1` 和 `set2` 分别变成了什么?
- 执行完 `set1.clear()` 方法之后, 集合 `set1` 变成了什么?

21.7 给出下面代码的输出结果:

```
import java.util.*;

public class Test {
    public static void main(String[] args) {
        LinkedHashSet<String> set1 = new LinkedHashSet<>();
        set1.add("New York");
        LinkedHashSet<String> set2 = set1;
        LinkedHashSet<String> set3 =
            (LinkedHashSet<String>)(set1.clone());
        set1.add("Atlanta");
        System.out.println("set1 is " + set1);
        System.out.println("set2 is " + set2);
        System.out.println("set3 is " + set3);
    }
}
```

21.8 给出下面代码的输出结果:

```
import java.util.*;
import java.io.*;


public class Test {
    public static void main(String[] args) throws Exception {
        ObjectOutputStream output = new ObjectOutputStream(
            new FileOutputStream("c:\\test.dat"));
        LinkedHashSet<String> set1 = new LinkedHashSet<>();
        set1.add("New York");
        LinkedHashSet<String> set2 =
            (LinkedHashSet<String>)set1.clone();
        set1.add("Atlanta");
        output.writeObject(set1);
        output.writeObject(set2);
        output.close();

        ObjectInputStream input = new ObjectInputStream(
            new FileInputStream("c:\\test.dat"));
        set1 = (LinkedHashSet<String>)input.readObject();
        set2 = (LinkedHashSet<String>)input.readObject();
        System.out.println(set1);
        System.out.println(set2);
        input.close();
    }
}
```

21.9 如果程序清单 21-5 中的第 6 ~ 7 行被下面的代码所替换, 输出将会是什么?

```
Set<GeometricObject> set = new HashSet<>();
```

21.3 比较集合和线性表的性能

 **要点提示:** 在无重复元素进行排序方面, 集合比线性表更加高效。线性表在通过索引来访问元素方面非常有用。

线性表中的元素可以通过索引来访问。而集合不支持索引, 因为集合中的元素是无序的。要遍历集合中的所有元素, 使用 `foreach` 循环。现在, 我们来做一个有趣的试验, 测试

集合和线性表的性能。程序清单 21-6 给出一个程序，该程序显示了（1）测试一个元素是否在一个散列集、链式散列集、树形集、数组线性表以及链表中，以及（2）从一个散列集、链式散列集、树形集、数组线性表以及链表中删除元素的执行时间。

程序清单 21-6 SetListPerformanceTest.java

```

1  import java.util.*;
2
3  public class SetListPerformanceTest {
4      static final int N = 50000;
5
6      public static void main(String[] args) {
7          // Add numbers 0, 1, 2, ..., N - 1 to the array list
8          List<Integer> list = new ArrayList<>();
9          for (int i = 0; i < N; i++)
10             list.add(i);
11          Collections.shuffle(list); // Shuffle the array list
12
13          // Create a hash set, and test its performance
14          Collection<Integer> set1 = new HashSet<>(list);
15          System.out.println("Member test time for hash set is " +
16              getTestTime(set1) + " milliseconds");
17          System.out.println("Remove element time for hash set is " +
18              getRemoveTime(set1) + " milliseconds");
19
20          // Create a linked hash set, and test its performance
21          Collection<Integer> set2 = new LinkedHashSet<>(list);
22          System.out.println("Member test time for linked hash set is " +
23              getTestTime(set2) + " milliseconds");
24          System.out.println("Remove element time for linked hash set is " +
25              + getRemoveTime(set2) + " milliseconds");
26
27          // Create a tree set, and test its performance
28          Collection<Integer> set3 = new TreeSet<>(list);
29          System.out.println("Member test time for tree set is " +
30              getTestTime(set3) + " milliseconds");
31          System.out.println("Remove element time for tree set is " +
32              getRemoveTime(set3) + " milliseconds");
33
34          // Create an array list, and test its performance
35          Collection<Integer> list1 = new ArrayList<>(list);
36          System.out.println("Member test time for array list is " +
37              getTestTime(list1) + " milliseconds");
38          System.out.println("Remove element time for array list is " +
39              getRemoveTime(list1) + " milliseconds");
40
41          // Create a linked list, and test its performance
42          Collection<Integer> list2 = new LinkedList<>(list);
43          System.out.println("Member test time for linked list is " +
44              getTestTime(list2) + " milliseconds");
45          System.out.println("Remove element time for linked list is " +
46              getRemoveTime(list2) + " milliseconds");
47      }
48
49      public static long getTestTime(Collection<Integer> c) {
50          long startTime = System.currentTimeMillis();
51
52          // Test if a number is in the collection
53          for (int i = 0; i < N; i++)
54              c.contains((int)(Math.random() * 2 * N));
55
56          return System.currentTimeMillis() - startTime;

```

```
57     }
58
59     public static long getRemoveTime(Collection<Integer> c) {
60         long startTime = System.currentTimeMillis();
61
62         for (int i = 0; i < N; i++)
63             c.remove(i);
64
65         return System.currentTimeMillis() - startTime;
66     }
67 }
```

```
Member test time for hash set is 20 milliseconds
Remove element time for hash set is 27 milliseconds
Member test time for linked hash set is 27 milliseconds
Remove element time for linked hash set is 26 milliseconds
Member test time for tree set is 47 milliseconds
Remove element time for tree set is 34 milliseconds
Member test time for array list is 39802 milliseconds
Remove element time for array list is 16196 milliseconds
Member test time for linked list is 52197 milliseconds
Remove element time for linked list is 14870 milliseconds
```

程序创建了一个包含数字 0 到 $N-1$ ($N=50\,000$) 的线性表 (第 8 ~ 10 行), 并打乱线性表 (第 11 行)。程序然后基于这个线性表创建一个散列集 (第 14 行)、一个链式散列集 (第 21 行)、一个树形集 (第 28 行)、一个数组线性表 (第 35 行) 以及一个链表 (第 42 行)。该程序获得测试一个数字是否在散列集中 (第 16 行)、链式散列集中 (第 23 行)、树形集中 (第 30 行)、数组线性表中 (第 37 行) 以及链表中 (第 44 行) 的执行时间; 然后获得将一个元素从散列集中 (第 18 行)、链式散列集中 (第 25 行)、树形集中 (第 32 行)、数组线性表中 (第 39 行) 以及链表中 (第 46 行) 删除的执行时间。

`getTestTime` 方法调用 `contains` 方法测试一个数字是否在容器中 (第 54 行), `getRemoveTime` 方法调用 `remove` 方法将一个元素从容器中移除 (第 63 行)。

如这些运行时间所展示的, 在测试一个元素是否在集合或者线性表的方面, 集合比线性表更加高效。因此, 前述的禁飞名单应该使用集合实现, 而不要采用线性表, 因为测试一个元素是否在一个集合中比测试它是否在一个线性表中要快得多。

你可能困惑为什么集合比线性表要更加高效。这些问题将在第 24 章和第 27 章介绍线性表和集合的实现的时候得到回答。

✓ 复习题

- 21.10 假定你需要编写一个无序存储无重复元素的程序, 应该使用什么数据结构?
- 21.11 假定你需要编写一个按照插入顺序来存储无重复元素的程序, 应该使用什么数据结构?
- 21.12 假定你需要编写一个以元素值升序存储无重复元素的程序, 应该使用什么数据结构?
- 21.13 假定你需要编写一个存储固定个数元素 (可能有重复元素) 的程序, 应该使用什么数据结构?
- 21.14 假定你需要编写一个程序, 将元素存储在一个线性表中并且需要经常在线性表的末尾进行添加和删除元素的操作, 应该使用什么数据结构?
- 21.15 假定你需要编写一个程序, 将元素存储在一个线性表中并且需要经常在线性表的开始处进行插入和删除元素的操作, 应该使用什么数据结构?

21.4 示例学习: 统计关键字

🔑 要点提示: 本节给出一个程序, 对一个 Java 源文件中的关键字进行计数。

对于 Java 源文件中的每个单词，需要确定该单词是否是一个关键字。为了高效处理这个问题，将所有关键字保存在一个 HashSet 中，并且使用 contains 方法来测试一个单词是否在关键字集合中。程序清单 21-7 给出了这个程序。

程序清单 21-7 CountKeywords.java

```

1  import java.util.*;
2  import java.io.*;
3
4  public class CountKeywords {
5      public static void main(String[] args) throws Exception {
6          Scanner input = new Scanner(System.in);
7          System.out.print("Enter a Java source file: ");
8          String filename = input.nextLine();
9
10         File file = new File(filename);
11         if (file.exists()) {
12             System.out.println("The number of keywords in " + filename
13                 + " is " + countKeywords(file));
14         }
15         else {
16             System.out.println("File " + filename + " does not exist");
17         }
18     }
19
20     public static int countKeywords(File file) throws Exception {
21         // Array of all Java keywords + true, false and null
22         String[] keywordString = {"abstract", "assert", "boolean",
23             "break", "byte", "case", "catch", "char", "class", "const",
24             "continue", "default", "do", "double", "else", "enum",
25             "extends", "for", "final", "finally", "float", "goto",
26             "if", "implements", "import", "instanceof", "int",
27             "interface", "long", "native", "new", "package", "private",
28             "protected", "public", "return", "short", "static",
29             "strictfp", "super", "switch", "synchronized", "this",
30             "throw", "throws", "transient", "try", "void", "volatile",
31             "while", "true", "false", "null"};
32
33         Set<String> keywordSet =
34             new HashSet<>(Arrays.asList(keywordString));
35         int count = 0;
36
37         Scanner input = new Scanner(file);
38
39         while (input.hasNext()) {
40             String word = input.next();
41             if (keywordSet.contains(word))
42                 count++;
43         }
44
45         return count;
46     }
47 }

```

Enter a Java source file: c:\Welcome.java
 The number of keywords in c:\Welcome.java is 5

Enter a Java source file: c:\TTT.java
 File c:\TTT.java does not exist

程序提示用户输入一个 Java 源文件（第 7 行）并且读取文件名（第 8 行）。如果文件存在，则调用 countKeywords 方法来统计文件中出现的关键字（第 13 行）。

countKeywords 方法创建了一个所有关键字的字符串数组（第 22 ~ 31 行），并且从该数组创建一个散列集合（第 33 ~ 34 行）。然后从文件中读取每个单词，并且测试这个单词是否在集合中（第 41 行）。如果在，程序增加 1 个计数（第 42 行）。

也可以使用 `LinkedHashSet`、`TreeSet`、`ArrayList` 或者 `LinkedList` 来存储关键字。然而，对这个程序来说，使用 `HashSet` 是最高效的。

✓ 复习题

21.16 如果第 33 ~ 34 行改为以下语句，CountKeywords 程序还能工作么？

```
Set<String> keywordSet =
    new LinkedHashSet<>(Arrays.asList(keywordString));
```

21.17 如果第 33 ~ 34 行改为以下语句，CountKeywords 程序还能工作么？

```
List<String> keywordSet =
    new ArrayList<>(Arrays.asList(keywordString));
```

21.5 映射表

🔑 要点提示：可以使用三个具体的类来创建一个映射表：`HashMap`、`LinkedHashMap`、`TreeMap`。

映射表（map）是一种依照键 / 值对存储元素的容器。它提供了通过键快速获取、删除和更新键 / 值对的功能。映射表将值和键一起保存。键很像下标。在 `List` 中，下标是整数；而在 `Map` 中，键可以是任意类型的对象。映射表中不能有重复的键，每个键都对应一个值。一个键和它的对应值构成一个条目并保存在映射表中，如图 21-2a 所示。图 21-2b 展示了一个映射表，其中每个条目由作为键的社会安全号以及作为值的姓名所组成。

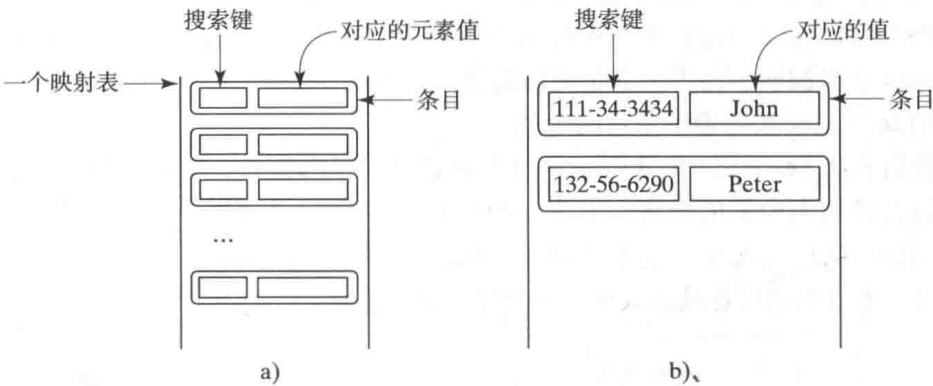


图 21-2 由键 / 值对组成的条目存储在映射表中

图的类型有三种：散列映射表 `HashMap`、链式散列映射表 `LinkedHashMap` 和树形映射表 `TreeMap`。这些映射表的通用特性都定义在 `Map` 接口中，它们的关系如图 21-3 所示。



图 21-3 映射表存储的是键 / 值对

Map 接口提供了查询、更新和获取合集的值和合集的键的方法，如图 21-4 所示。

«interface» <i>java.util.Map<K,V></i>	
<pre> +clear(): void +containsKey(key: Object): boolean +containsValue(value: Object): boolean +entrySet(): Set<Map.Entry<K,V>> +get(key: Object): V +isEmpty(): boolean +keySet(): Set<K> +put(key: K, value: V): V +putAll(m: Map<? extends K,? extends V>): void +remove(key: Object): V +size(): int +values(): Collection<V> </pre>	<p>从该映射表中删除所有条目 如果该映射表包含了指定键的条目，则返回 true</p> <p>如果该映射表将一个或者多个键映射到指定值，则返回 true</p> <p>返回一个包含了该映射表中条目的集合 返回该映射表中指定键对应的值 如果该映射表中没有包含任何条目，则返回 true 返回一个包含该映射表中所有键的集合 将一个条目放入该映射表中 将 m 中的所有条目添加到该映射表中</p> <p>删除指定键对应的条目 返回该映射表中的条目数 返回该映射表中所有值组成的合集</p>

图 21-4 Map 接口将键映射到值

更新方法 (update method) 包括 clear、put、putAll 和 remove。方法 clear() 从映射表中删除所有的条目。方法 put(K key,V value) 为映射表中指定的键和值添加条目。如果这个映射表原来就包含该键的一个条目，则原来的值将被新的值所替代，并且返回与这个键相关联的原来的值。方法 putAll(Map m) 将 m 中的所有条目添加到这个映射表中。方法 remove(Object key) 将指定键对应的条目从映射表中删除。

查询方法 (query method) 包括 containsKey、containsValue、isEmpty 和 size。方法 containsKey(Object key) 检测映射表中是否包含指定键的条目。方法 containsValue(Object value) 检测图中是否包含指定值的条目。方法 isEmpty() 检测映射表中是否包含条目。方法 size() 返回映射表中条目的个数。

可以使用方法 keySet() 来获得一个包含映射表中键的集合，也可以使用方法 values() 获得一个包含映射表中值的合集。方法 entrySet() 返回一个所有条目的集合。这些条目是 Map.Entry<K,V> 接口的实例，这里 Entry 是 Map 接口的一个内部接口，如图 21-5 所示。该集合中的每个条目都是所在映射表中一个特定的键 / 值对。

«interface» <i>java.util.Map.Entry<K,V></i>	
<pre> +getKey(): K +getValue(): V +setValue(value: V): void </pre>	<p>返回该条目的键 返回该条目的值 将该条目中的值赋以新的值</p>

图 21-5 Map.Entry 接口在映射表中的条目上操作

AbstractMap 类是一个便利抽象类，它实现了 Map 接口中除了 entrySet() 方法之外的所有方法。

HashMap、LinkedHashMap 和 TreeMap 类是 Map 接口的三个具体实现 (concrete implementation)，如图 21-6 所示。

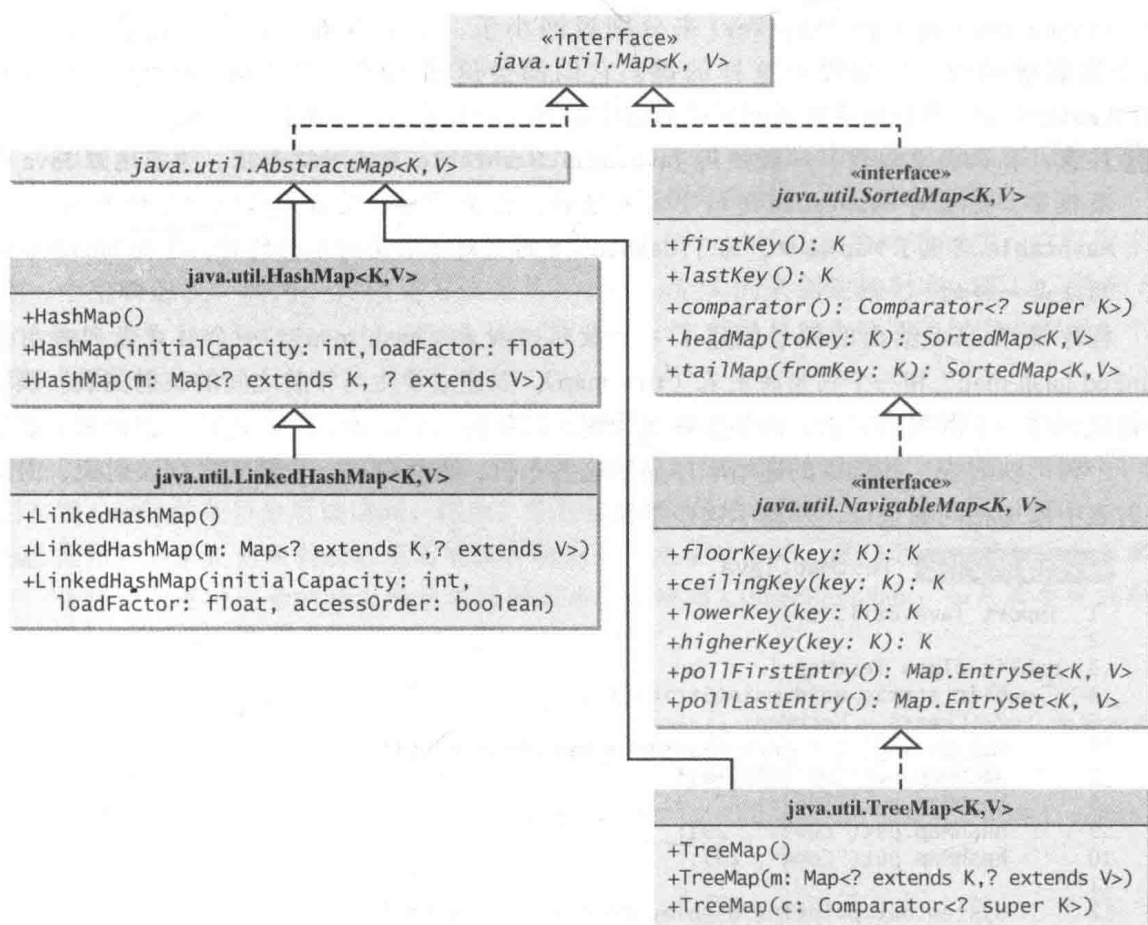


图 21-6 Java 合集框架提供三个具体映射表类

对于定位一个值、插入一个条目以及删除一个条目而言，HashMap 类是高效的。

LinkedHashMap 类用链表实现来扩展 HashMap 类，它支持映射表中条目的排序。HashMap 类中的条目是没有顺序的，但是在 LinkedHashMap 中，元素既可以按照它们插入映射表的顺序排序（称为插入顺序（insertion order）），也可以按它们被最后一次访问时的顺序，从最早到最晚（称为访问顺序（access order））排序。无参构造方法是以插入顺序来创建 LinkedHashMap 对象的。要按访问顺序创建 LinkedHashMap 对象，应该使用构造方法 `LinkedHashMap (initialCapacity, loadFactor, true)`。

TreeMap 类在遍历排好顺序的键时是很高效的。键可以使用 Comparable 接口或 Comparator 接口来排序。如果使用它的无参构造方法创建一个 TreeMap 对象，假定键的类实现了 Comparable 接口，则可以使用 Comparable 接口中的 `compareTo` 方法来对映射表内的键进行比较。要使用比较器，必须使用构造方法 `TreeMap(Comparator comparator)` 来创建一个有序映射表，这样，该映射表中的条目就能使用比较器中的 `compare` 方法按键进行排序。

SortedMap 是 Map 的一个子接口，使用它可确保映射表中的条目是排好序的。除此之外，它还提供方法 `firstKey()` 和 `lastKey()` 来返回映射表中的第一个和最后一个键，而方法 `headMap(toKey)` 和 `tailMap(fromKey)` 分别返回键小于 `toKey` 的那部分映射表和键大于或等于 `fromKey` 的那部分映射表。

NavigableMap 继承了 SortedMap，以提供导航方法 `lowerKey(key)`、`floorKey(key)`、

`ceilingKey(key)` 和 `higherKey(key)` 来分别返回小于、小于或等于、大于或等于、大于某个给定键的键，如果没有这样的键，它们都会返回 `null`。方法 `pollFirstEntry()` 和 `pollLastEntry()` 分别删除并返回树映射表中的第一个和最后一个条目。

{} 注意：在 Java 2 以前，一般使用 `java.util.Hashtable` 来映射键和值。为了适应 Java 集合框架，Java 对 `Hashtable` 进行了重新设计，但是，为了向后兼容保留了所有的方法。`Hashtable` 实现了 `Map` 接口，除了 `Hashtable` 的更新方法是同步的的外，它与 `HashMap` 的用法是一样的。

程序清单 21-8 给出的例子创建了一个散列映射表 (`hash msp`)、一个链式散列映射表 (`linked hash map`) 和一个树形映射表 (`tree map`)，以建立学生与年龄之间的映射关系。该程序首先创建一个散列映射表，以学生姓名为键，以年龄为它的值，然后由这个散列映射表创建一个树形映射表，并按键的递增顺序显示这些条目，最后创建一个链式散列映射表，向该映射表中添加相同的条目，并显示这些条目。

程序清单 21-8 TestMap.java

```

1  import java.util.*;
2
3  public class TestMap {
4      public static void main(String[] args) {
5          // Create a HashMap
6          Map<String, Integer> hashMap = new HashMap<>();
7          hashMap.put("Smith", 30);
8          hashMap.put("Anderson", 31);
9          hashMap.put("Lewis", 29);
10         hashMap.put("Cook", 29);
11
12         System.out.println("Display entries in HashMap");
13         System.out.println(hashMap + "\n");
14
15         // Create a TreeMap from the preceding HashMap
16         Map<String, Integer> treeMap =
17             new TreeMap<>(hashMap);
18         System.out.println("Display entries in ascending order of key");
19         System.out.println(treeMap);
20
21         // Create a LinkedHashMap
22         Map<String, Integer> linkedHashMap =
23             new LinkedHashMap<>(16, 0.75f, true);
24         linkedHashMap.put("Smith", 30);
25         linkedHashMap.put("Anderson", 31);
26         linkedHashMap.put("Lewis", 29);
27         linkedHashMap.put("Cook", 29);
28
29         // Display the age for Lewis
30         System.out.println("\nThe age for " + "Lewis is " +
31             linkedHashMap.get("Lewis"));
32
33         System.out.println("Display entries in LinkedHashMap");
34         System.out.println(linkedHashMap);
35     }
36 }

```

Display entries in HashMap
{Cook=29, Smith=30, Lewis=29, Anderson=31}

Display entries in ascending order of key
{Anderson=31, Cook=29, Lewis=29, Smith=30}


```
The age for Lewis is 29
Display entries in LinkedHashMap
{Smith=30, Anderson=31, Cook=29, Lewis=29}
```

如输出所示, `HashMap` 中条目的顺序是随机的, 而 `TreeMap` 中的条目是按键的升序排列的, `LinkedHashMap` 中的条目则是按元素最后一次被访问的时间从早到晚排序的。

实现 `Map` 接口的所有具体类至少有两种构造方法: 一种是无参构造方法, 它可用来创建一个空映射表, 而另一种构造方法是从 `Map` 的一个实例来创建映射表。所以, 语句 `new TreeMap<String,Integer>(hashMap)` (第 16 ~ 17 行) 就是从一个散列映射表来创建一个树形映射表。

可以创建一个按插入顺序或访问顺序排序的链式散列映射表。程序第 22 ~ 23 行创建一个按访问顺序排序的链式散列映射表, 最晚被访问的条目被放在映射表的末尾。在第 31 行, 拥有键 `Lewis` 的条目最后被访问, 因此, 它在第 34 行最后被显示。

提示: 如果更新映射表时不需要保持映射表中元素的顺序, 就使用 `HashMap`; 如果需要保持映射表中元素的插入顺序或访问顺序, 就使用 `LinkedHashMap`; 如果需要使映射表按照键排序, 就使用 `TreeMap`。

✓ 复习题

- 21.18 如何创建 `Map` 的一个实例? 如何向由键和值组成的映射表中添加一个条目? 如何从映射表中删除一个条目? 如何获取映射表的大小? 如何遍历映射表中的条目?
- 21.19 描述并比较散列映射表 `HashMap`、链式散列映射表 `LinkedHashMap` 和树形映射表 `TreeMap`。
- 21.20 给出下面代码的输出结果:

```
public class Test {
    public static void main(String[] args) {
        Map<String, String> map = new LinkedHashMap<>();
        map.put("123", "John Smith");
        map.put("111", "George Smith");
        map.put("123", "Steve Yao");
        map.put("222", "Steve Yao");
        System.out.println("(1) " + map);
        System.out.println("(2) " + new TreeMap<String, String>(map));
    }
}
```

21.6 示例学习: 单词的出现次数

要点提示: 该示例学习编写一个程序, 以统计一个文本中单词的出现次数, 然后按照单词的字母顺序显示这些单词以及它们对应的出现次数。

本程序使用一个 `TreeMap` 来存储包含单词及其次数的条目。对于每一个单词来说, 都要判断它是否已经是映射表中的一个键。如果不是, 将由这个单词为键而 1 为值构成的条目存入该映射表中。否则, 将映射表中该单词 (键) 对应的值加 1。假定单词是不区分大小写的, 例如, `Good` 被认为是和 `good` 一样的。

程序清单 21-9 给出了该问题的解决方法。

程序清单 21-9 CountOccurrenceOfWords.java

```
1 import java.util.*;
2
3 public class CountOccurrenceOfWords {
```

```

4 public static void main(String[] args) {
5     // Set text in a string
6     String text = "Good morning. Have a good class. " +
7         "Have a good visit. Have fun!";
8
9     // Create a TreeMap to hold words as key and count as value
10    Map<String, Integer> map = new TreeMap<>();
11
12    String[] words = text.split("[ \\n\\t\\r.,;:!?@{}]");
13    for (int i = 0; i < words.length; i++) {
14        String key = words[i].toLowerCase();
15
16        if (key.length() > 0) {
17            if (!map.containsKey(key)) {
18                map.put(key, 1);
19            }
20            else {
21                int value = map.get(key);
22                value++;
23                map.put(key, value);
24            }
25        }
26    }
27
28    // Get all entries into a set
29    Set<Map.Entry<String, Integer>> entrySet = map.entrySet();
30
31    // Get key and value from each entry
32    for (Map.Entry<String, Integer> entry: entrySet)
33        System.out.println(entry.getKey() + "\\t" + entry.getValue());
34    }
35 }

```

a	2
class	1
fun	1
good	3
have	3
morning	1
visit	1

该程序创建了一个 `TreeMap` (第 10 行) 来存储包含单词和它的出现次数的条目。单词被看做是键。因为映射表中的所有值必须存储为对象, 所以统计次数被包装在一个 `Integer` 对象中。

程序使用 `String` 类 (参见 10.10.4 节) 中的 `split` 方法 (第 12 行) 从文本中提取单词。对于每个被提取出的单词, 程序都会检测它是否已经被存储为映射表中的键 (第 17 行)。如果没有, 就将这个单词和它的初始统计次数 (1) 构成一个新条目, 然后存储到映射表中 (第 18 行)。否则, 给该单词的计数器加 1 (第 21 ~ 23 行)。

程序获取集合中映射表的条目 (第 29 行), 然后遍历这个集合以显示每个条目中的统计次数和键 (第 32 ~ 33 行)。

因为这个映射表是一个树形映射表, 所以条目是以单词的升序显示的。要以出现次数的升序显示它们, 参见编程练习题 21.8。

现在回过头思考一下, 在不使用映射表的情况下如何编写这个程序。新程序将会更长, 也更复杂, 由此可发现映射表是解决此类问题的非常高效且功能强大的数据结构。

✓ 复习题

21.21 如果第 10 行改成下面语句, 程序 `CountOccurrenceOfWords` 还能工作吗?

```
Map<String, int> map = new TreeMap<>();
```

21.22 如果第 17 行改成下面语句，程序 CountOccurrenceOfWords 还能工作吗？

```
if (map.get(key) == null) {
```

21.23 如果第 32 ~ 33 行改成下面语句，程序 CountOccurrenceOfWords 还能工作吗？

```
for (String key: map)
    System.out.println(key + "\t" + map.getValue(key));
```

21.7 单元素与不可变的合集和映射表

🔑 **要点提示：**可以使用 Collections 类中的静态方法来创建单元素的集合、线性表和映射表，以及不可变集合、线性表和映射表。

Collections 类包含了用于线性表和合集的静态方法。它还包含用于创建不可修改的单元素的集合、线性表和映射表的方法，以及用于创建只读集合、线性表和映射表的方法，如图 21-7 所示。

java.util.Collections	
+singleton(o: Object): Set	返回一个包含了指定对象的不可修改的集合
+singletonList(o: Object): List	返回一个包含了指定对象的不可修改的线性表
+singletonMap(key: Object, value: Object): Map	返回一个具有键值对的不可修改的映射表
+unmodifiableCollection(c: Collection): Collection	返回一个合集的只读视图
+unmodifiableList(list: List): List	返回一个线性表的只读视图
+unmodifiableMap(m: Map): Map	返回一个映射表的只读视图
+unmodifiableSet(s: Set): Set	返回一个集合的只读视图
+unmodifiableSortedMap(s: SortedMap): SortedMap	返回一个排好序的映射表的只读视图
+unmodifiableSortedSet(s: SortedSet): SortedSet	返回一个排好序的集合的只读视图

图 21-7 Collections 类包含了用于创建单元素并且只读的集合、线性表和映射表的静态方法

Collections 类中定义了三个常量：一个表示空的集合，一个表示空线性表，一个表示空映射表 (EMPTY_SET、EMPTY_LIST 和 EMPTY_MAP)。这些合集是不可修改的。该类中还定义了如下几个方法：方法 singleton(Object o) 用于创建仅含一个条目的不可变集合；方法 singletonList(Object o) 用于创建仅含一个条目的不可变线性表；方法 singletonMap(Object key, Object value) 用于创建仅含一个单一条目的不可变映射表。

Collections 类还提供了 6 个用于返回合集的只读视图的静态方法：unmodifiableCollection(Collection c)、unmodifiableList(List list)、unmodifiableMap(Map m)、unmodifiableSet(Set set)、unmodifiableSortedMap(SortedMap m) 和 unmodifiableSortedSet(SortedSet s)。这种类型的视图类似于真正合集的引用。但是不能通过一个只读的视图来修改合集。尝试通过只读视图修改合集将引发 UnsupportedOperationException 异常。

✓ 复习题

21.24 下面代码中有什么错误？

```
Set<String> set = Collections.singleton("Chicago");
set.add("Dallas");
```

21.2 运行下面代码的时候将发生什么？

```
List list = Collections.unmodifiableList(Arrays.asList("Chicago",
    "Boston"));
list.remove("Dallas");
```

关键术语

hash map (散列映射表)	set (集合)
hash set (散列集)	read-only view(只读视图)
linked hash map (链式散列映射表)	tree map (树形映射表)
linked hash set (键式散列集)	tree set (树形集)
map (映射表)	

本章小结

1. 集合存储的是不重复的元素。若要在合集中存储重复的元素,需要使用线性表。
2. 映射表中存储的是键/值对。它提供使用键快速查询一个值。
3. Java 合集框架支持三种类型的集合:散列集 `HashSet`、链式散列集 `LinkedHashSet` 和树形集 `TreeSet`。`HashSet` 以一个不可预知的顺序存储元素;`LinkedHashSet` 以元素被插入的顺序存储元素;`TreeSet` 存储已排好序的元素。`HashSet`、`LinkedHashSet` 和 `TreeSet` 中的所有方法都继承自 `Collection` 接口。
4. `Map` 接口将键映射到元素上。键类似于索引。`List` 中,索引为整数。`Map` 中,键可以为任何对象。映射表不能包含相同的键。每个键可以映射最多一个值。`Map` 接口提供了查询、更新以及获取值的合集以及键的合集的方法。
5. Java 合集框架支持三种类型的映射表:散列映射表 `HashMap`、链式散列映射表 `LinkedHashMap` 和树形映射表 `TreeMap`。对于定位一个值、插入一个条目和删除一个条目而言,`HashMap` 是很高效的。`LinkedHashMap` 支持映射表中的条目排序。`HashMap` 类中的条目是没有顺序的,但 `LinkedHashMap` 中的条目可以按某种顺序来获取,该顺序既可以是它们被插入映射表中的顺序(称为插入顺序),也可以是它们最后一次被访问的时间的顺序,从最早到最晚(称为访问顺序)。对于遍历排好序的键,`TreeMap` 是高效的。键可以使用 `Comparable` 接口来排序,也可以使用 `Comparator` 接口来排序。

测试题

回答位于网址 www.cs.armstrong.edu/liang/intro10e/quiz.html 的本章测试题。

编程练习题

21.2 ~ 21.4 节

- 21.1 (在散列集上进行集合操作) 创建两个链接散列集合 `{"George","Jim","John","Blake","Kevin","Michael"}` 和 `{"George","Katie","Kevin","Michelle","Ryan"}`, 然后求它们的并集、差集和交集。(可以先备份一份这些集合,以防随后进行的集合操作改变原来的集合。)
- 21.2 (按升序显示不重复的单词) 编写一个程序,从文本文件中读取单词,并将所有不重复的单词按升序显示。文本文件被作为命令行参数传递。
- **21.3 (统计 Java 源代码中的关键字) 修改程序清单 21-7 中的程序。如果关键字在注释或者字符串中,则不进行统计。将 Java 文件名从命令行传递。假设 Java 源代码是正确的,行注释和段落注释不会交叉。
- *21.4 (统计元音和辅音) 编写一个程序,提示用户输入一个文本文件名,然后显示文件中的元音和辅音的数目。使用一个集合存储元音 A、E、I、O 和 U。
- ***21.5 (突出显示语法) 编写一个程序,将一个 Java 文件转换为一个 HTML 文件。在 HTML 文件中,关键字、注释和字面量分别用粗体的深蓝色、绿色和蓝色显示。使用命令行传递 Java 文件和 HTML 文件。例如,下面的命令

java Exercise21_05 Welcome.java Welcome.html

将 Welcome.java 转换为 Welcome.html。图 21-8a 显示了一个 Java 文件，它对应的 HTML 文件如图 21-8b 所示。

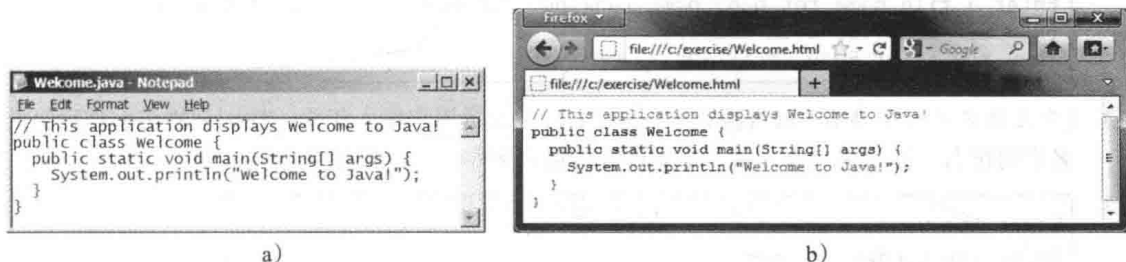


图 21-8 a 中纯文本形式的 Java 代码被显示在 b 中的 HTML 中，其中突出了它的语法

21.5 ~ 21.7 节

- *21.6 (统计输入数字的个数) 编写一个程序，读取个数不定的整数，然后查找其中出现频率最高的数字。当输入为 0 时，表示结束输入。例如，如果输入的数据是 2 3 40 3 5 4 -3 3 3 2 0，那么数字 3 的出现频率是最高的。如果出现频率最高的数字不是一个而是多个，则应该将它们全部报告。例如，在线性表 9 30 3 9 3 2 4 中，3 和 9 都出现了两次，所以 3 和 9 都应该被报告。
- **21.7 (改写程序清单 21-9) 改写程序清单 21-9，将单词按出现频率的升序显示。
(提示：创建一个名为 WordOccurrence 的类实现 Comparable 接口。这个类包含两个域：word 和 count。使用 compareTo 方法比较单词的出现次数。对程序清单 21-9 散列集中的每个对，创建 WordOccurrence 的一个实例，并把它储存到一个数组线性表中。使用 Collections.sort 方法对该数组线性表进行排序。如果将 WordOccurrence 的实例存入树形集，会发生什么错误？)
- **21.8 (统计文本文件中单词的出现频率) 改写程序清单 21-9，从文本文件中读取文本，文本文件名被作为命令行参数传递。单词由空格、标点符号(,;.:?)、引号('") 以及括号分隔。统计单词不区分大小写(例如，认为 Good 和 good 是一样的单词)。单词必须以字母开头。以单词的字母顺序显示输出，每个单词前面显示它的出现次数。
- **21.9 (使用映射表猜首府) 改写编程练习题 8.37，在映射表中存储州和它的首府的条目。你的程序应该提示用户输入一个州，然后显示这个州的首府。
- *21.10 (统计每个关键字的出现次数) 重写程序清单 21-7，读入一个 Java 源代码文件并且统计文件中每个关键字的出现次数。如果关键字是在注释中或者字符串面值中，则不要进行统计。
- **21.11 (婴儿姓名流行度排名) 使用编程练习题 12.31 中的数据文件编写一个程序，使得用户可以选择一个年份、性别，输入一个姓名，然后显示在选择的年份和性别条件下，该姓名的排名，如图 21-9 所示。为了获得最好的效率，为男孩名字和女孩名字分别创建两个数组。每个数组针对 10 个年份具有 10 个元素。每个元素是一个映射表，以值对的方式存储了姓名和相应的排名，并将姓名作为键。假设数据文件保存在 www.cs.armstrong.edu/liang/data/babynamesranking2001.txt, ..., www.cs.armstrong.edu/liang/data/babynamesranking2010.txt 中。

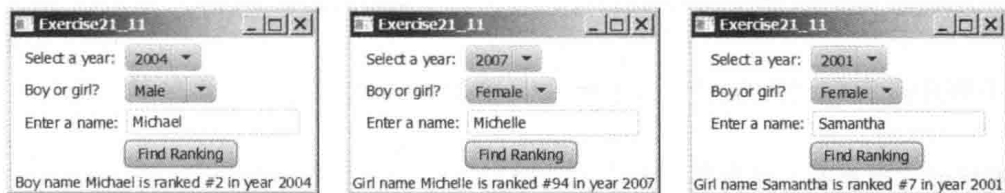


图 21-9 用户选择一个年份和性别，输入年份，单击 Find Ranking 按钮显示排名

- **21.12** (可以同时用于两个性别的姓名) 编写一个程序, 提示用户输入编程练习题 12.31 中描述的文件名, 然后显示文件中可以同时用于两种性别的姓名。使用集合存储姓名并找到两个集合中的共同姓名。下面是一个运行示例:

```
Enter a file name for baby name ranking: babynamesranking2001.txt   
69 names used for both genders  
They are Tyler Ryan Christian ...
```

- **21.13** (婴儿姓名流行度排名) 修改编程练习题 21.11, 提示用户输入年份、性别和姓名, 然后显示该名字的排名。提示用户输入另一个查询或者退出程序。下面是一个运行示例:

```
Enter the year: 2010   
Enter the gender: M   
Enter the name: Javier   
Boy name Javier is ranked #190 in year 2010  
Enter another inquiry? Y   
Enter the year: 2001   
Enter the gender: F   
Enter the name: Emily   
Girl name Emily is ranked #1 in year 2001  
Enter another inquiry? N 
```

- **21.14** (Web 爬虫) 重写编程练习题 12.18, 为 `ListOfPendingURLs` 和 `listofTraversedURLs` 采用合适的新的数据结构以提高性能。
- **21.15** (加法测试题) 重写编程练习题 11.16, 将答案保存在一个集合中, 而不是线性表中。

开发高效算法

【】 教学目标

- 使用符号大 O 估计算法效率 (22.2 节)。
- 理解增长率以及为什么在估计时可以忽略常量和非主导项 (22.2 节)。
- 确定各种类型算法的复杂度 (22.3 节)。
- 分析二分查找算法 (22.4.1 节)。
- 分析选择排序算法 (22.4.2 节)。
- 分析汉诺塔算法 (22.4.3 节)。
- 描述常用的增长型函数 (常量时间、对数时间、对数-线性时间、二次时间、三次时间和指数时间) (22.4.4 节)。
- 使用动态编程设计找到斐波那契数的高效算法 (22.5 节)。
- 使用欧几里得算法找到最大公约数 (22.6 节)。
- 使用埃拉托色尼筛选法找到素数 (22.7 节)。
- 使用分而治之的方法设计找到最近距离点对的高效算法 (22.8 节)。
- 使用回溯法解决八皇后问题 (22.9 节)。
- 设计高效算法, 为一个点集找到凸包 (22.10 节)。

22.1 引言

🔑 要点提示: 算法设计是为了解决某个问题开发一个数学流程。算法分析是预测一个算法的性能。

前面两章介绍了经典的数据结构 (线性表、栈、队列、优先队列、集合和映射表), 并将它们应用于解决问题。本章将采用各种示例来介绍如何用通用的算法技术 (动态编程、分而治之以及回溯) 来开发高效的算法。本书稍后的第 23 ~ 29 章将介绍一些高效的算法。在介绍高效算法的开发之前, 我们需要讨论关于如何衡量算法效率的问题。

22.2 使用大 O 符号来衡量算法效率

🔑 要点提示: 大 O 符号标记可以基于输入的大小得到一种衡量算法的时间复杂度的函数。可以忽略函数中的倍乘常量和非主导项。

假定两个算法执行相同的任务, 比如查找 (线性查找与二分查找), 哪个算法更好呢? 为了回答这个问题, 我们可以实现这两个算法, 并运行程序得到执行时间。但是这种方法存在以下两个问题:

- 首先, 计算机上同时运行着许多任务, 一个特定程序的执行时间是依赖于系统负荷的。
- 其次, 执行时间依赖于特定的输入。例如, 考虑线性查找和二分查找。如果要查找的

元素恰巧是线性表中的第一个元素，那么线性查找会比二分查找更快找到该元素。

通过测量它们的执行时间来比较算法是非常困难的。为了克服这些问题，计算机科学家开发了一个独立于计算机和指定输入的理论方法来分析算法。该方法大致估计了由输入大小的改变而产生的影响。通过这个方法可以看到随着输入大小的增长算法执行时间增长得有多快，因此可以通过检查两个算法的增长率 (growth rate) 来比较它们。

考虑线性查找的问题。线性查找算法顺序比较数组中的元素与键，直到找到键或者数组已搜索完毕。如果该键不在数组中，那么对于一个大小为 n 的数组需要 n 次比较。如果该键在数组中，那么平均需要 $n/2$ 次比较。该算法的执行时间与数组的大小成正比。如果将数组大小加倍，那么比较次数也会加倍。该算法是呈线性增长的，增长率是 n 的数量级。计算机科学家使用大 O 符号 (Big O notation) 表示数量级。使用该符号，线性查找算法的复杂度就是 $O(n)$ ，读为“ n 阶”。我们将时间复杂度为 $O(n)$ 的算法称为线性算法，它体现为线性的增长率。

对于相同的输入大小，算法的执行时间可能会随着输入的不同而不同。导致最短执行时间的输入称为最佳情况输入 (best-case input)；而导致最长执行时间的输入称为最差情况输入 (worst-case input)。最佳情况分析和最差情况分析用来分析最佳情况输入和最差情况输入的算法。最佳和最差情况分析都不具有代表性，但是最差情况分析却是非常有用的。我们可以确定的是自己的算法永远不会比最差情况还慢。平均情况分析 (average-case analysis) 试图在所有可能的相同大小的输入中确定平均时间。平均情况分析是比较理想的，但是很难完成，这是因为对于许多问题而言，要确定各种输入实例的相对概率和分布是相当困难的。由于最差情况分析比较容易完成，所以分析通常针对最差情况进行。

如果你几乎总是在线性表中查找一个已知道存在于其中的元素，那么线性查找算法在最差情况下需要 n 次比较，而在平均情况下需要 $n/2$ 次比较。使用大 O 符号，这两种情况需要的时间都为 $O(n)$ 。倍乘常量 ($1/2$) 可以忽略。算法分析的重点在于增长率，而倍乘常量对增长率没有影响。对于 $n/2$ 或 $100n$ 而言，增长率都和 n 一样，如表 22-1 所示。因此， $O(n)=O(n/2)=O(100n)$ 。

表 22-1 增长率

$f(n)$	n	$n/2$	$100n$	
n	n	$n/2$	$100n$	
100	100	50	10000	
200	200	100	20000	
	2	2	2	$f(200)/f(100)$

考虑在包含 n 个元素的数组中找出最大数的算法。如果 n 为 2，找到最大数需要一次比较；如果 n 为 3，找到最大数需要两次比较。一般来说，在拥有 n 个元素的线性表中找到最大数需要 $n-1$ 次比较。算法分析主要用于庞大的输入规模。如果输入规模较小，那么估计算法效率是没有意义的。随着 n 的增大，表达式 $n-1$ 中的 n 就主导了复杂度。大 O 符号允许忽略非主导部分 (例如，表达式 $n-1$ 中的 -1)，并强调重要部分 (例如，表达式 $n-1$ 中的 n)。因此，该算法的复杂度为 $O(n)$ 。

大 O 标记估算一个算法与输入规模相关的执行时间。如果执行时间与输入规模无关，就称该算法耗费了常量时间 (constant time)，用符号 $O(1)$ 表示。例如，在数组中从给定下

标处获取元素的方法耗费的时间即为常量时间，这是因为该时间不会随数组规模的增大而增长。

在算法分析中经常会用到下面的数学求和公式：

$$1+2+3+\dots+(n-2)+(n-1)=\frac{n(n-1)}{2}=O(n^2)$$

$$1+2+3+\dots+(n-2)+n=\frac{n(n+1)}{2}=O(n^2)$$

$$a^0+a^1+a^2+a^3+\dots+a^{(n-1)}+a^n=\frac{a^{n+1}-1}{a-1}=O(a^n)$$

$$2^0+2^1+2^2+2^3+\dots+2^{(n-1)}+2^n=\frac{2^{n+1}-1}{2-1}=2^{n+1}-1=O(2^n)$$

注意：时间复杂度是使用大 O 标记对运行时间进行测量。类似的，也可以使用大 O 标记对空间复杂度进行测量。空间复杂度是使用算法测量内存空间的大小。本书中大多数算法的空间复杂度为 $O(n)$ 。即，相对于输入问题大小，它们体现出线性的增长率。例如，线性查找的空间复杂度为 $O(n)$ 。

复习题

22.1 为什么大 O 标记中忽略掉常量因子？为什么大 O 标记中忽略掉非主导项？

22.2 下面各个函数分别为多少阶？

$$\frac{(n^2+1)^2}{n}, \frac{(n^2+\log^2 n)^2}{n}, n^3+100n^2+n, 2^n+100n^2+45n, n2^n+n^22^n$$

22.3 示例：确定大 O

要点提示：本节给出多个示例，为循环、顺序以及选择语句确定大 O 。

示例 1

考虑下面循环的时间复杂度：

```
for (int i = 1; i <= n; i++) {
    k = k + 5;
}
```

执行下面的语句是一个常量时间 c ，

```
k = k + 5;
```

因为循环执行了 n 次，因此其时间复杂度是

$$T(n) = (\text{a constant } c) * n = O(n)$$

理论分析预测了算法的性能。为了观察这个算法的执行，运行程序清单 22-1 中的代码来获得 $n = 1\,000\,000$, $10\,000\,000$, $100\,000\,000$ 以及 $1\,000\,000\,000$ 的运行时间。

程序清单 22-1 PerformanceTest.java

```
1 public class PerformanceTest {
2     public static void main(String[] args) {
3         getTime(1000000);
4         getTime(10000000);
5         getTime(100000000);
6         getTime(1000000000);
7     }
8 }
```

```

7   }
8
9   public static void getTime (long n) {
10      long startTime = System.currentTimeMillis();
11      long k = 0;
12      for (int i = 1; i <= n; i++) {
13         k = k + 5;
14      }
15      long endTime = System.currentTimeMillis();
16      System.out.println("Execution time for n = " + n
17         + " is " + (endTime - startTime) + " milliseconds");
18   }
19 }

```

```

Execution time for n = 1000000 is 6 milliseconds
Execution time for n = 10000000 is 61 milliseconds
Execution time for n = 100000000 is 610 milliseconds
Execution time for n = 1000000000 is 6048 milliseconds

```

我们前面的分析预测了这个循环的线性时间复杂度。如示例运行所显示的，当输入问题的大小增加了 10 倍，运行时间也增加了大约 10 倍。运行和预测是吻合的。

示例 2

下面循环的时间复杂度是多少？

```

for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        k = k + i + j;
    }
}

```

执行下面的语句是一个常量时间 c

```
k = k + i + j;
```

外层循环执行 n 次。外层循环的每次迭代，内层循环都会执行 n 次。因此，该循环的时间复杂度是

$$T(n) = (\text{a constant } c) \times n \times n = O(n^2)$$

时间复杂度为 $O(n^2)$ 的算法称为平方级算法 (quadratic algorithm)，表现为平方级增长率。平方级算法随着问题规模的增加快速增长。如果输入规模加倍，算法时间就变成 4 倍。通常，两层嵌套循环的算法都是平方级的。

示例 3

考虑下面的循环：

```

for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        k = k + i + j;
    }
}

```

外层循环执行 n 次。对于 $i=1, 2, \dots$ ，内层循环分别执行 1 次、2 次以及 n 次。因此，该循环的时间复杂度是

$$\begin{aligned}
 T(n) &= c + 2c + 3c + 4c + \dots + nc \\
 &= cn(n+1)/2 \\
 &= (c/2)n^2 + (c/2)n \\
 &= O(n^2)
 \end{aligned}$$

示例 4

考虑下面的循环：

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= 20; j++) {
        k = k + i + j;
    }
}
```

内层循环执行 20 次，外层循环执行 n 次。因此，该循环的时间复杂度是

$$T(n) = 20 \times c \times n = O(n)$$

示例 5

考虑下面的语句：

```
for (int j = 1; j <= 10; j++) {
    k = k + 4;
}
```

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= 20; j++) {
        k = k + i + j;
    }
}
```

第一个循环执行 10 次，第二个循环执行 $20 \times n$ 次。因此，该循环的时间复杂度是

$$T(n) = 10 \times c + 20 \times c \times n = O(n)$$

示例 6

考虑下面的选择语句：

```
if (list.contains(e)) {
    System.out.println(e);
}
else
    for (Object t: list) {
        System.out.println(t);
    }
```

假设线性表中包含了 n 个元素，那么 `list.contains(e)` 的执行时间是 $O(n)$ 。在 `else` 子句中的循环耗费的时间是 $O(n)$ 。因此，整个语句的时间复杂度是

$$\begin{aligned} T(n) &= \text{if test time} + \text{worst-case time (if clause, else clause)} \\ &= O(n) + O(n) = O(n) \end{aligned}$$

示例 7

考虑计算 a^n ， n 为整数。一个简单的算法就是将 a 乘 n 次，如下所示：

```
result = 1;
for (int i = 1; i <= n; i++)
    result *= a;
```

这个算法耗费的时间是 $O(n)$ 。不失一般性，假设 $n=2^k$ 。可以使用下面的方案提高算法的效率：

```
result = a;
for (int i = 1; i <= k; i++)
    result = result * result;
```

这个算法耗费的时间是 $O(\log n)$ 。可以修改算法以针对任意的 n ，并证明它的复杂度仍然是 $O(\log n)$ 。（参见复习题 22.7。）

{ } 注意: 为了简单起见, 因为 $O(\log n) = O(\log_2 n) = O(\log_a n)$, 所以常量的底可忽略。

✓ 复习题

22.3 下列循环中的循环次数是多少?

```
int count = 1;
while (count < 30) {
    count = count * 2;
}
```

a)

```
int count = 15;
while (count < 30) {
    count = count * 3;
}
```

b)

```
int count = 1;
while (count < n) {
    count = count * 2;
}
```

c)

```
int count = 15;
while (count < n) {
    count = count * 3;
}
```

d)

22.4 如果 n 为 10, 那么下面的代码将显示多少个星号? 如果 n 为 20, 将显示多少个星号? 使用大 O 标记估算时间复杂度。

```
for (int i = 0; i < n; i++) {
    System.out.print('*');
}
```

a)

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        System.out.print('*');
    }
}
```

b)

```
for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            System.out.print('*');
        }
    }
}
```

c)

```
for (int k = 0; k < 10; k++) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            System.out.print('*');
        }
    }
}
```

d)

22.5 使用符号 O 估算下列方法的时间复杂度。

```
public static void mA(int n) {
    for (int i = 0; i < n; i++) {
        System.out.print(Math.random());
    }
}
```

a)

```
public static void mB(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            System.out.print(Math.random());
        }
    }
}
```

b)

```
public static void mC(int[] m) {
    for (int i = 0; i < m.length; i++) {
        System.out.print(m[i]);
    }

    for (int i = m.length - 1; i >= 0; i--) {
        System.out.print(m[i]);
    }
}
```

c)


```
public static void mD(int[] m) {
    for (int i = 0; i < m.length; i++) {
        for (int j = 0; j < i; j++) {
            System.out.print(m[i] * m[j]);
        }
    }
}
```

d)

22.6 设计一个 $O(n)$ 时间的算法, 计算从 n_1 到 n_2 的数字的和 ($n_1 < n_2$)。可以设计一个 $O(1)$ 复杂度的算法来计算同样的任务吗?

22.7 22.3 节的示例 7 假设 $n = 2^k$ 。针对任意的 n 改进算法, 证明复杂度仍然为 $O(\log n)$ 。

22.4 分析算法的时间复杂度

 **要点提示:** 本节将分析几种著名算法的复杂度, 这些算法包括: 二分查找法、选择排序法和汉诺塔法。

22.4.1 分析二分查找算法

在程序清单 7-7 中给出的二分查找算法, 是在一个有序数组中查找一个键。算法中的每次迭代都包含固定次数的操作, 次数由 c 来表示。设 $T(n)$ 表示在包含 n 个元素的线性表中进行二分查找的时间复杂度。不失一般性, 假定 n 是 2 的幂, 且 $k = \log n$ 。在两次比较之后, 二分查找排除了输入的一半,

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{2^2}\right) + c + c = T\left(\frac{n}{2^k}\right) + kc \\ &= T(1) + c \log n = 1 + (\log n)c \\ &= O(\log n) \end{aligned}$$

忽略常量和非主导项, 二分查找算法的复杂度为 $O(\log n)$ 。具有 $O(\log n)$ 时间复杂度的算法称为对数算法 (logarithmic algorithm), 体现了对数级的增长率。 \log 的底为 2, 但是底不会影响对数增长率, 因此可以将其忽略。随着问题规模的增长, 对数算法复杂度增长得比较缓慢。在二分查找的示例中, 将数组的大小翻倍, 最多增加一次的比较。如果输入规模平方, 那么算法的时间复杂度只会加倍。因此, 对数-时间算法是很高效的。

22.4.2 分析选择排序算法

在程序清单 7-8 中给出的选择排序算法, 是在线性表中找到最小元素, 并将其和第一个元素交换。然后在剩下的元素中找到最小元素, 将其和剩余的线性表中的第一个元素交换, 这样一直做下去, 直到线性表中仅剩一个元素为止。对于第一次迭代, 比较次数为 $n-1$; 第二次迭代的比较次数为 $n-2$, 以此类推。设 $T(n)$ 表示选择排序的复杂度, c 表示每次迭代中其他操作的总数, 如赋值和附加的比较。这样,

$$\begin{aligned} T(n) &= (n-1) + c + (n-2) + c + \cdots + 2 + c + 1 + c \\ &= \frac{(n-1)(n-1+1)}{2} + c(n-1) = \frac{n^2}{2} - \frac{n}{2} + cn - c \\ &= O(n^2) \end{aligned}$$

因此, 选择排序算法的复杂度为 $O(n^2)$ 。

22.4.3 分析汉诺塔问题

在程序清单 18-8 中给出的汉诺塔问题, 按如下方式借助塔 C 将 n 个盘子从塔 A 递归地移动到塔 B:

- 1) 借助塔 B 将前 $n-1$ 个盘子从塔 A 移动到塔 C。
- 2) 将盘子 n 从塔 A 移动到塔 B。
- 3) 借助塔 A 将 $n-1$ 个盘子从塔 C 移动到塔 B。

这个算法的复杂度由移动的次数来衡量。设 $T(n)$ 表示使用该算法从塔 A 到塔 B 移动 n 个盘子需要的移动次数, $T(1)$ 为 1。因此,

$$\begin{aligned} T(n) &= T(n-1) + 1 + T(n-1) \\ &= 2T(n-1) + 1 \\ &= 2(2T(n-2) + 1) + 1 \\ &= 2(2(2T(n-3) + 1) + 1) + 1 \\ &= 2^{n-1}T(1) + 2^{n-2} + \cdots + 2 + 1 \\ &= 2^{n-1} + 2^{n-2} + \cdots + 2 + 1 = (2^n - 1) = O(2^n) \end{aligned}$$

具有 $O(2^n)$ 时间复杂度的算法称为指数算法 (exponential algorithm), 体现为指数级的增长率。随着输入规模的增长, 指数算法耗费的时间呈指数增长。指数算法在庞大的输入规模下并不实用。假设盘子一次移动耗时一秒, 则需要耗费 $2^{32}/(365 \times 24 \times 60 \times 60) = 136$ 年来移动 32 个盘子, 以及 $2^{64}/(365 \times 24 \times 60 \times 60) = 5850$ 亿年来移动 64 个盘子。

22.4.4 常用的递推关系

递推关系 (Recurrence relation) 对于分析算法的复杂度非常有用。如前面例子所示, 二分查找、选择排序以及汉诺塔问题的复杂度分别为 $T(n) = T\left(\frac{n}{2}\right) + O(1)$, $T(n) = T(n-1) + O(n)$, $T(n) = 2T(n-1) + O(1)$ 。表 22-2 总结了常用的递推关系。

表 22-2 常用的递推关系

递推关系	结果	示例
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$	二分查找, 欧几里得法求最大公约数
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$	线性查找
$T(n) = 2T(n/2) + O(1)$	$T(n) = O(n)$	复习题 22.20
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$	归并排序 (23 章)
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$	选择排序
$T(n) = 2T(n-1) + O(1)$	$T(n) = O(2^n)$	汉诺塔
$T(n) = T(n-1) + T(n-2) + O(1)$	$T(n) = O(2^n)$	递归的斐波那契算法

22.4.5 比较常用的增长函数

前面几节分析了几个算法的复杂度。表 22-3 列出了一些常用的增长函数, 然后显示当输入规模从 $n = 25$ 加倍到 $n = 50$ 时, 增长率是如何变化的。

表 22-3 增长率的变化

函数	名称	$n = 25$	$n = 50$	$f(50)/f(25)$
$O(1)$	常量时间	1	1	1
$O(\log n)$	对数时间	4.64	5.64	1.21
$O(n)$	线性时间	25	50	2
$O(n \log n)$	对数 - 线性时间	116	282	2.43
$O(n^2)$	二次时间	625	2500	4
$O(n^3)$	三次时间	15625	125000	8
$O(2^n)$	指数时间	3.36×10^7	1.27×10^{15}	3.35×10^7

这些函数的比较关系如下, 变化趋势如图 22-1 所示。

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

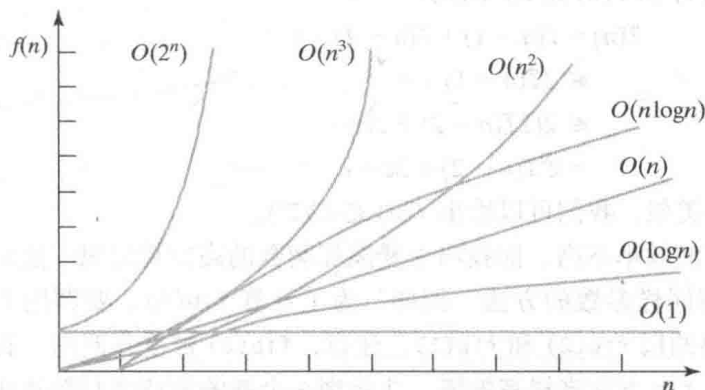


图 22-1 随着规模 n 的增大, 函数的增长趋势

✓ 复习题

22.8 依序排列下面的增长函数:

$$\frac{5n^3}{4032}, 44\log n, 10n\log n, 500, 2n^2, \frac{2^n}{45}, 3n$$

22.9 估算将两个 $n \times m$ 矩阵相加的时间复杂度, 以及将 $n \times m$ 矩阵与 $m \times k$ 矩阵相乘的时间复杂度。

22.10 描述寻找数组中最大元素出现次数的算法。分析该算法的复杂度。

22.11 描述从数组中删除重复元素的算法。分析该算法的复杂度。

22.12 分析下面的排序算法:

```
for (int i = 0; i < list.length - 1; i++) {
    if (list[i] > list[i + 1]) {
        swap list[i] with list[i + 1];
        i = -1;
    }
}
```

22.13 分析分别使用穷举法和 Horner 方法计算对于给定 x 值的 n 阶多项式 $f(x)$ 的复杂度。穷举法通过计算多项式中的每项并将其相加。Horner 方法在 6.7 节介绍过。

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \dots + a_1 x^1 + a_0$$

22.5 使用动态编程计算斐波那契数

🔑 要点提示: 本节使用动态编程为计算斐波那契数分析和设计一个高效算法。

本书 18.3 节给出了一个找出斐波那契数的递归方法, 如下所示:

```
/** The method for finding the Fibonacci number */
public static long fib(Long index) {
    if (index == 0) // Base case
        return 0;
    else if (index == 1) // Base case
        return 1;
    else // Reduction and recursive calls
        return fib(index - 1) + fib(index - 2);
}
```

现在，我们可以证明这个算法的复杂度是 $O(2^n)$ 。为了方便起见，令下标为 n 。假设 $T(n)$ 表示找出 $\text{fib}(n)$ 的算法的复杂度，而 c 表示比较下标为 0 的数和下标为 1 的数耗费的常量时间，也就是说 $T(1)$ 和 $T(0)$ 是 c 。因此，

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + c \\ &\leq 2T(n-1) + c \\ &\leq 2(2T(n-2) + c) + c \\ &= 2^2T(n-2) + 2c + c \end{aligned}$$

和汉诺塔问题的分析类似，我们可以给出 $T(n)$ 是 $O(2^n)$ 。

然而，这个算法的效率不高。能找出求斐波那契数的高效算法吗？递归的 fib 方法中的问题在于冗余地调用同样参数的方法。例如，为了计算 $\text{fib}(4)$ ，要调用 $\text{fib}(3)$ 和 $\text{fib}(2)$ 。为了计算 $\text{fib}(3)$ ，要调用 $\text{fib}(2)$ 和 $\text{fib}(1)$ 。注意， $\text{fib}(2)$ 被重复调用。我们可以通过避免重复调用同样参数的 fib 方法来提高效率。注意到一个新的斐波那契数是通过数列中的前两个数相加得到的。如果用两个变量 $f0$ 和 $f1$ 来存储前面的两个数，那么可以通过将 $f0$ 和 $f1$ 相加立即获得新数 $f2$ 。现在，应该通过将 $f1$ 赋给 $f0$ ，将 $f2$ 赋给 $f1$ 来更新 $f0$ 和 $f1$ ，如图 22-2 所示。

	f0	f1	f2											
斐波那契数列:	0	1	1	2	3	5	8	13	21	34	55	89	...	
索引:	0	1	2	3	4	5	6	7	8	9	10	11		

	f0	f1	f2											
斐波那契数列:	0	1	1	2	3	5	8	13	21	34	55	89	...	
索引:	0	1	2	3	4	5	6	7	8	9	10	11		

										f0	f1	f2		
斐波那契数列:	0	1	1	2	3	5	8	13	21	34	55	89	...	
索引:	0	1	2	3	4	5	6	7	8	9	10	11		

图 22-2 变量 $f0$ 、 $f1$ 和 $f2$ 存储数列中三个连续斐波那契数

新的方法在程序清单 22-2 中实现。

程序清单 22-2 ImprovedFibonacci.java

```

1 import java.util.Scanner;
2
3 public class ImprovedFibonacci {
4     /** Main method */
5     public static void main(String args[]) {
6         // Create a Scanner
7         Scanner input = new Scanner(System.in);
8         System.out.print("Enter an index for the Fibonacci number: ");
9         int index = input.nextInt();
10
11         // Find and display the Fibonacci number
12         System.out.println(
13             "Fibonacci number at index " + index + " is " + fib(index));
14     }
15
16     /** The method for finding the Fibonacci number */
17     public static long fib(long n) {
18         long f0 = 0; // For fib(0)
19         long f1 = 1; // For fib(1)
20         long f2 = 1; // For fib(2)
21     }

```

```
22     if (n == 0)
23         return f0;
24     else if (n == 1)
25         return f1;
26     else if (n == 2)
27         return f2;
28
29     for (int i = 3; i <= n; i++) {
30         f0 = f1;
31         f1 = f2;
32         f2 = f0 + f1;
33     }
34
35     return f2;
36 }
37 }
```

Enter an index for the Fibonacci number: 6
Fibonacci number at index 6 is 8

Enter an index for the Fibonacci number: 7
Fibonacci number at index 7 is 13

很显然，新算法的复杂度是 $O(n)$ ，比递归的 $O(2^n)$ 算法提高了不少。

这里给出的计算斐波那契数的算法使用了一种称为动态编程（dynamic programming）的方法。动态编程是通过解决子问题，然后将子问题的结果结合来获得整个问题的解的过程。这自然地引向递归的解答。然而，使用递归将效率不高，因为子问题相互重叠了。动态编程的关键思想是只解决子问题一次，并将子问题的结果存储以备后用，从而避免了重复的子问题的求解。

✓ 复习题

22.14 什么是动态编程？给出一个动态编程的示例。

22.15 为什么递归的斐波那契算法是低效的，而非递归的斐波那契算法是高效的？

22.6 使用欧几里得算法求最大公约数

🔑 要点提示：本节给出几个求两个整数最大公约数的算法，以在其中找出一个高效的算法。

两个整数的最大公约数（greatest common divisor, GCD）是能被这两个整数整除的最大数。程序清单 5-9 给出了一个求两个整数 m 和 n 的最大公约数的穷举算法。穷举（brute force）法指使用最简单和直接，或者非常明显的方式解决问题的一种算法。结果是，为了解决一个给定问题，这样的算法相比更聪明或者更复杂的算法而言，可能导致做更多的工作。另外一方面，穷举算法相对于复杂的算法而言，通常更加易于实现，并且因为其简单性，有时候可以更加高效。

穷举算法检测 k ($k=2, 3, 4, \dots$) 是否是 n_1 和 n_2 的公约数，直到 k 大于 n_1 或 n_2 。该算法可以如下描述：

```
public static int gcd(int m, int n) {
    int gcd = 1;

    for (int k = 2; k <= m && k <= n; k++) {
        if (m % k == 0 && n % k == 0)
            gcd = k;
    }
}
```

```
    return gcd;
}
```

假设 $m \geq n$, 那么, 显然该算法的复杂度是 $O(n)$ 。

是否还有求最大公约数的更好的算法? 不从 1 向上开始查找可能的除数, 而是从 n 开始向下查找, 这样会更高效。一旦找到一个除数, 该除数就是最大公约数。因此, 可以使用下面的循环来改进算法:

```
for (int k = n; k >= 1; k--) {
    if (m % k == 0 && n % k == 0) {
        gcd = k;
        break;
    }
}
```

这个算法比前一个效率更高, 但是它的最坏情况的时间复杂度依旧是 $O(n)$ 。

数字 n 的除数不可能比 $n/2$ 大。因此, 可以使用下面的循环进一步改进算法:

```
for (int k = m / 2; k >= 1; k--) {
    if (m % k == 0 && n % k == 0) {
        gcd = k;
        break;
    }
}
```

但是, 该算法是不正确的, 因为 n 可能会是 m 的除数。这种情况必须考虑到。正确的算法如程序清单 22-3 所示。

程序清单 22-3 GCD.java

```
1 import java.util.Scanner;
2
3 public class GCD {
4     /** Find GCD for integers m and n */
5     public static int gcd(int m, int n) {
6         int gcd = 1;
7
8         if (m % n == 0) return n;
9
10        for (int k = n / 2; k >= 1; k--) {
11            if (m % k == 0 && n % k == 0) {
12                gcd = k;
13                break;
14            }
15        }
16
17        return gcd;
18    }
19
20    /** Main method */
21    public static void main(String[] args) {
22        // Create a Scanner
23        Scanner input = new Scanner(System.in);
24
25        // Prompt the user to enter two integers
26        System.out.print("Enter first integer: ");
27        int m = input.nextInt();
28        System.out.print("Enter second integer: ");
29        int n = input.nextInt();
30
31        System.out.println("The greatest common divisor for " + m +
32            " and " + n + " is " + gcd(m, n));
33    }
34 }
```

```
33 }
34 }
```

```
Enter first integer: 2525
Enter second integer: 125
The greatest common divisor for 2525 and 125 is 25
```

```
Enter first integer: 3
Enter second integer: 3
The greatest common divisor for 3 and 3 is 3
```

假设 $m \geq n$ ，那么这个 for 循环最多执行 $n/2$ 次，比前一个算法节省了一半的运行时间。该算法的时间复杂度仍然是 $O(n)$ ，但实际上，它比程序清单 5-9 中的算法快得多。

注意：大 O 标记提供了对算法效率的一个很好的理论上的估算。但是，两个算法即使有相同的时间复杂度，它们的效率也不一定相同。如前面的例子所示，程序清单 5-9 和程序清单 22-3 中的两个算法具有相同的复杂度，但实际上，程序清单 22-3 中的算法显然更好些。

求最大公约数的一个更有效的算法是在公元前 300 年左右由欧几里得发现的，这是最古老的著名算法之一。它可以递归地定义如下：

用 $\text{gcd}(m, n)$ 表示整数 m 和 n 的最大公约数：

- 如果 $m \% n$ 为 0，那么 $\text{gcd}(m, n)$ 为 n 。
- 否则， $\text{gcd}(m, n)$ 就是 $\text{gcd}(n, m \% n)$ 。

不难证明这个算法的正确性。假设 $m \% n = r$ ，那么， $m = qn + r$ ，这里的 q 是 m/n 的商。能整除 m 和 n 的任意数字都必须也能整除 r 。因此， $\text{gcd}(m, n)$ 和 $\text{gcd}(n, r)$ 是一样的，其中 $r = m \% n$ 。该算法的实现如程序清单 22-4 所示。

程序清单 22-4 GCDEuclid.java

```
1 import java.util.Scanner;
2
3 public class GCDEuclid {
4     /** Find GCD for integers m and n */
5     public static int gcd(int m, int n) {
6         if (m % n == 0)
7             return n;
8         else
9             return gcd(n, m % n);
10    }
11
12    /** Main method */
13    public static void main(String[] args) {
14        // Create a Scanner
15        Scanner input = new Scanner(System.in);
16
17        // Prompt the user to enter two integers
18        System.out.print("Enter first integer: ");
19        int m = input.nextInt();
20        System.out.print("Enter second integer: ");
21        int n = input.nextInt();
22
23        System.out.println("The greatest common divisor for " + m +
24            " and " + n + " is " + gcd(m, n));
25    }
26 }
```

```
Enter first integer: 2525 
Enter second integer: 125 
The greatest common divisor for 2525 and 125 is 25
```

```
Enter first integer: 3 
Enter second integer: 3 
The greatest common divisor for 3 and 3 is 3
```

最好的情况是当 $m \% n$ 为 0 的时候，算法只用一步就能找出最大公约数。分析平均情况是很困难的。然而，我们可以证明最坏情况的时间复杂度是 $O(\log n)$ 。

假设 $m \geq n$ ，我们可以证明 $m\%n < m/2$ ，如下所示：

- 如果 $n \leq m/2$ ，那么 $m\%n < m/2$ ，因为 m 除以 n 的余数总是小于 n 。
- 如果 $n > m/2$ ，那么 $m\%n = m - n < m/2$ 。因此， $m\%n < m/2$ 。

欧几里得的算法递归地调用 gcd 方法。它首先调用 $\text{gcd}(m, n)$ ，接着调用 $\text{gcd}(n, m\%n)$ ，然后是 $\text{gcd}(m\%n, n\%(m\%n))$ ，以此类推，如下所示：

```
gcd(m, n)
= gcd(n, m % n)
= gcd(m % n, n % (m % n))
= ...
```

因为 $m\%n < m/2$ 且 $n\%(m\%n) < n/2$ ，所以传递给 gcd 方法的参数在每两次迭代之后减少一半。在调用 gcd 两次之后，第二个参数小于 $n/2$ 。在调用 gcd 四次之后，第二个参数小于 $n/4$ 。在调用 gcd 六次之后，第二个参数小于 $n/2^3$ 。假设 k 是调用 gcd 方法的次数。在调用 gcd 方法 k 次之后，第二个参数小于 $n/2^{(k/2)}$ ，它是大于或等于 1 的。也就是

$$\frac{n}{2^{(k/2)}} \geq 1 \Rightarrow n \geq 2^{(k/2)} \Rightarrow \log n \geq k/2 \Rightarrow k \leq 2 \log n$$

因此， $k \leq 2 \log n$ 。所以该 gcd 方法的时间复杂度是 $O(\log n)$ 。

最坏情况发生在两个数导致了最大分离的时候，两个连续的斐波那契数会造成最大分离的情况。回顾斐波那契数列是从 0 和 1 开始，然后后一个数都是前两个数的和，例如：

0 1 1 2 3 5 8 13 21 34 55 89...

这个数列可以递归地定义为

```
fib(0) = 0;
fib(1) = 1;
fib(index) = fib(index - 2) + fib(index - 1); index >= 2
```

对于两个连续的斐波那契数 $\text{fib}(\text{index})$ 和 $\text{fib}(\text{index}-1)$ ，

```
gcd(fib(index), fib(index - 1))
= gcd(fib(index - 1), fib(index - 2))
= gcd(fib(index - 2), fib(index - 3))
= gcd(fib(index - 3), fib(index - 4))
= ...
= gcd(fib(2), fib(1))
= 1
```

例如，

```
gcd(21, 13)
= gcd(13, 8)
= gcd(8, 5)
= gcd(5, 3)
= gcd(3, 2)
= gcd(2, 1)
= 1
```

因此，gcd 方法被调用的次数和下标相等。我们可以证明 $\text{index} \leq 1.44 \log n$ ，其中 $n = \text{fib}(\text{index} - 1)$ 。这是一个比 $\text{index} \leq 2 \log n$ 更严格的限定。

表 22-4 总结了三个求最大公约数的算法的复杂度。

表 22-4 GCD 算法的比较

算法	复杂度	描述
程序清单 5-9	$O(n)$	穷举法，检查所有可能的除数
程序清单 22-3	$O(n)$	检查所有可能除数的一半
程序清单 22-4	$O(\log n)$	欧几里得算法

复习题

22.16 证明下面寻找两个整数 m 和 n 的 GCD 的算法是错误的。

```
int gcd = 1;
for (int k = Math.min(Math.sqrt(n), Math.sqrt(m)); k >= 1; k--) {
    if (m % k == 0 && n % k == 0) {
        gcd = k;
        break;
    }
}
```

22.7 寻找素数的高效算法

要点提示：本节给出了多个算法，以找到寻找素数的一个高效算法。

150 000 美元等待着第一个发现素数的个人或团体，这里的素数要求至少是 100 000 000 位十进制数字的数 (w2.eff.org/awards/coop-prime-rules.php)。

你可以设计一个寻找素数的快速算法吗？

对于一个大于 1 的整数，如果其除数只有 1 和它本身，那么它就是一个素数 (prime)。例如，2、3、5、7 都是素数，但是 4、6、8、9 都不是。

如何确定一个数字 n 是否是素数？程序清单 5-15 给出了一个求素数的穷举算法。算法检测 $2, 3, 4, 5, \dots, n-1$ 是否能整除 n 。如果不能，那么 n 就是素数。这个算法耗费 $O(n)$ 时间来检测 n 是否是一个素数。注意，只需要检测 $2, 3, 4, 5, \dots, n/2$ 是否能整除 n 。如果不能，那么 n 就是素数。算法的效率只稍微提高了一点，它的复杂度仍然是 $O(n)$ 。

实际上，我们可以证明，如果 n 不是素数，那么 n 必须有一个大于 1 且小于或等于 \sqrt{n} 的因子。下面是它的证明过程：因为 n 不是素数，所以会存在两个数 p 和 q ，满足 $n = pq$ 且 $1 < p \leq q$ 。注意， $n = \sqrt{n} \sqrt{n}$ 。 p 必须小于或等于 \sqrt{n} 。因此，只需要检测 $2, 3, 4, 5, \dots$ ，或者 \sqrt{n} 是否能被 n 整除。如果不能， n 就是素数。这会显著地降低时间复杂度，为 $O(\sqrt{n})$ 。

现在考虑找出不超过 n 的所有素数。一个直观的实现方法就是检测 i 是否是素数，这里 $i = 2, 3, 4, \dots, n$ 。程序在程序清单 22-5 中给出。

程序清单 22-5 PrimeNumbers.java

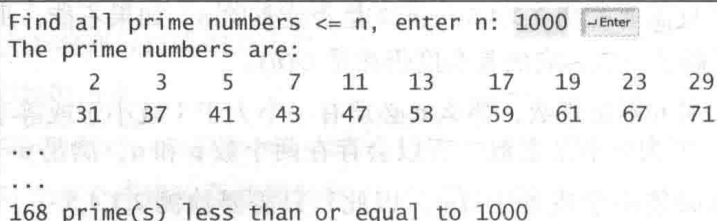
```
1 import java.util.Scanner;
2
3 public class PrimeNumbers {
4     public static void main(String[] args) {
5         Scanner input = new Scanner(System.in);
6         System.out.print("Find all prime numbers <= n, enter n: ");
7         int n = input.nextInt();
8     }
}
```



```

9    final int NUMBER_PER_LINE = 10; // Display 10 per line
10   int count = 0; // Count the number of prime numbers
11   int number = 2; // A number to be tested for primeness
12
13   System.out.println("The prime numbers are:");
14
15   // Repeatedly find prime numbers
16   while (number <= n) {
17       // Assume the number is prime
18       boolean isPrime = true; // Is the current number prime?
19
20       // Test if number is prime
21       for (int divisor = 2; divisor <= (int)(Math.sqrt(number));
22           divisor++) {
23           if (number % divisor == 0) { // If true, number is not prime
24               isPrime = false; // Set isPrime to false
25               break; // Exit the for loop
26           }
27       }
28
29       // Print the prime number and increase the count
30       if (isPrime) {
31           count++; // Increase the count
32
33           if (count % NUMBER_PER_LINE == 0) {
34               // Print the number and advance to the new line
35               System.out.printf("%7d\n", number);
36           }
37           else
38               System.out.printf("%7d", number);
39       }
40
41       // Check if the next number is prime
42       number++;
43   }
44
45   System.out.println("\n" + count +
46       " prime(s) less than or equal to " + n);
47 }
48 }

```



Find all prime numbers <= n, enter n: 1000

The prime numbers are:

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
...									
...									

168 prime(s) less than or equal to 1000

如果 for 循环的每次迭代都必须计算 `Math.sqrt(number)`，那么该程序的效率不高（第 21 行）。一个好的编译器应该为整个 for 循环只计算一次 `Math.sqrt(number)`。为确保出现这种情况，可以显式地用下面两行替换第 21 行：

```

int squareRoot = (int)(Math.sqrt(number));
for (int divisor = 2; divisor <= squareRoot; divisor++) {

```

实际上，没有必要对每个 number 来确切计算 `Math.sqrt(number)`。只需要找出完全平方数，例如，4、9、16、25、36、49，等等。注意，对于 36 和 48 之间并包括 36 和 48 的数，它们的 `(int)(Math.sqrt(number))` 为 6。认识到这一点，就可以用下面的代码替换第 16 ~ 26 行：

```

...
int squareRoot = 1;

// Repeatedly find prime numbers
while (number <= n) {
    // Assume the number is prime
    boolean isPrime = true; // Is the current number prime?

    if (squareRoot * squareRoot < number) squareRoot++;

    // Test if number is prime
    for (int divisor = 2; divisor <= squareRoot; divisor++) {
        if (number % divisor == 0) { // If true, number is not prime
            isPrime = false; // Set isPrime to false
            break; // Exit the for loop
        }
    }
}
...

```

现在，我们专注于分析该程序的复杂度上。因为它在 for 循环中耗费 \sqrt{i} 步（第 21 ~ 27 行）来检测数字 i 是否是素数，所以算法耗费 $\sqrt{2} + \sqrt{3} + \sqrt{4} + \dots + \sqrt{n}$ 步来找出所有小于或等于 n 的素数。注意到

$$\sqrt{2} + \sqrt{3} + \sqrt{4} + \dots + \sqrt{n} \leq n\sqrt{n}$$

因此，该算法的复杂度为 $O(n\sqrt{n})$ 。

为了确定 i 是否是素数，算法需要检测 2, 3, 4, 5, …，以及 \sqrt{i} 是否能被 i 整除。可以进一步提高该算法的效率，因为只需要检测从 2 到 \sqrt{i} 之间的素数能否被 i 整除。

我们可以证明，如果 i 不是素数，那就必须存在一个素数 p ，满足 $i=pq$ 且 $p \leq q$ 。下面是它的证明过程。假设 i 不是素数，且 p 是 i 的最小因子。那么 p 肯定是素数，否则， p 就有一个因子 k ，且 $2 \leq k < p$ 。 k 也是 i 的一个因子，这和 p 是 i 的最小因子是冲突的。因此，如果 i 不是素数，那么可以找出从 2 到 \sqrt{i} 之间的被 i 整除的素数。这会得到一个求不超过 n 的所有素数的更有效的算法，如程序清单 22-6 所示。

程序清单 22-6 EfficientPrimeNumbers.java

```

1  import java.util.Scanner;
2
3  public class EfficientPrimeNumbers {
4      public static void main(String[] args) {
5          Scanner input = new Scanner(System.in);
6          System.out.print("Find all prime numbers <= n, enter n: ");
7          int n = input.nextInt();
8
9          // A list to hold prime numbers
10         java.util.List<Integer> list =
11             new java.util.ArrayList<>();
12
13         final int NUMBER_PER_LINE = 10; // Display 10 per line
14         int count = 0; // Count the number of prime numbers
15         int number = 2; // A number to be tested for primeness
16         int squareRoot = 1; // Check whether number <= squareRoot
17
18         System.out.println("The prime numbers are \n");
19
20         // Repeatedly find prime numbers
21         while (number <= n) {
22             // Assume the number is prime
23             boolean isPrime = true; // Is the current number prime?
24
25             if (squareRoot * squareRoot < number) squareRoot++;

```

```

26
27 // Test whether number is prime
28 for (int k = 0; k < list.size()
29     && list.get(k) <= squareRoot; k++) {
30     if (number % list.get(k) == 0) { // If true, not prime
31         isPrime = false; // Set isPrime to false
32         break; // Exit the for loop
33     }
34 }
35
36 // Print the prime number and increase the count
37 if (isPrime) {
38     count++; // Increase the count
39     list.add(number); // Add a new prime to the list
40     if (count % NUMBER_PER_LINE == 0) {
41         // Print the number and advance to the new line
42         System.out.println(number);
43     }
44     else
45         System.out.print(number + " ");
46 }
47
48 // Check whether the next number is prime
49 number++;
50 }
51
52 System.out.println("\n" + count +
53 " prime(s) less than or equal to " + n);
54 }
55 }

```

Find all prime numbers <= n, enter n: 1000

The prime numbers are:

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
...									
...									

168 prime(s) less than or equal to 1000

假设 $\pi(i)$ 表示小于或等于 i 的素数的个数。20 以下的素数是 2、3、5、7、11、13、17 和 19。因此， $\pi(2)$ 是 1， $\pi(3)$ 是 2， $\pi(6)$ 是 3，而 $\pi(20)$ 是 8。已经证明过 $\pi(i)$ 近似为 $\frac{i}{\log i}$ (参见 primes.utm.edu/howmany.shtml)。

对每个数字 i ，该算法检测小于或等于 \sqrt{i} 的素数是否能被 i 整除。小于或等于 \sqrt{i} 的素数的个数是

$$\frac{\sqrt{i}}{\log \sqrt{i}} = \frac{2\sqrt{i}}{\log i}$$

这样，找出不超过 n 的所有素数的复杂度为

$$\frac{2\sqrt{2}}{\log 2} + \frac{2\sqrt{3}}{\log 3} + \frac{2\sqrt{4}}{\log 4} + \frac{2\sqrt{5}}{\log 5} + \frac{2\sqrt{6}}{\log 6} + \frac{2\sqrt{7}}{\log 7} + \frac{2\sqrt{8}}{\log 8} + \dots + \frac{2\sqrt{n}}{\log n}$$

对于 $i < n$ 且 $n \geq 16$ ，由于 $\frac{\sqrt{i}}{\log i} < \frac{\sqrt{n}}{\log n}$ ，

$$\frac{2\sqrt{2}}{\log 2} + \frac{2\sqrt{3}}{\log 3} + \frac{2\sqrt{4}}{\log 4} + \frac{2\sqrt{5}}{\log 5} + \frac{2\sqrt{6}}{\log 6} + \frac{2\sqrt{7}}{\log 7} + \frac{2\sqrt{8}}{\log 8} + \dots + \frac{2\sqrt{n}}{\log n} < \frac{2n\sqrt{n}}{\log n}$$

因此, 这个算法的复杂度为 $O\left(\frac{n\sqrt{n}}{\log n}\right)$ 。

这个算法是动态编程的另外一个示例。该算法在数组线性表中存储子问题的结果, 之后使用它们来检测一个新的数字是否是素数。

还有比 $O\left(\frac{n\sqrt{n}}{\log n}\right)$ 更好的算法吗? 让我们检测一下著名的找素数的埃拉托色尼算法。

Eratosthenes (公元前 276—194 年) 是一位希腊数学家, 他设计了一个称为埃拉托色尼筛选法 (sieve of Eratosthenes) 的聪明算法, 该算法求出所有小于或等于 n 的素数。该算法使用一个名为 `primes` 的数组, 其中有 n 个布尔值。初始状态时, `primes` 的所有元素都设置为 `true`。因为 2 的倍数都不是素数, 所以对于所有的 $2 \leq i < n/2$, 都将 `primes[2*i]` 设置为 `false`, 如图 22-3 所示。因为我们不关注 `primes[0]` 和 `primes[1]`, 所以这些值在图中被标注上 \times 。

素数数组

下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
初始值	\times	\times	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
$k=2$	\times	\times	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
$k=3$	\times	\times	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F
$k=5$	\times	\times	Ⓟ	Ⓟ	F	Ⓟ	F	Ⓟ	F	F	F	Ⓟ	F	Ⓟ	F	F	F	Ⓟ	F	Ⓟ	F	F	F	Ⓟ	F	F	F	F

图 22-3 `primes` 中的值随着每个素数 k 而改变

因为 3 的倍数不是素数, 所以对于所有的 $3 \leq i \leq n/3$, 都将 `primes[3*i]` 设置为 `false`。因为 5 的倍数不是素数, 所以对于所有的 $5 \leq i \leq n/5$, 都将 `primes[5*i]` 设置为 `false`。注意, 无须考虑 4 的倍数, 因为 4 的倍数也是 2 的倍数, 这已经考虑过了。同样, 6、8、9 的倍数也无须考虑。只需要考虑素数 $k=2,3,5,7,11,\dots$ 的倍数, 并且将 `primes` 中对应的元素设置为 `false`。之后, 如果 `primes[i]` 仍然为 `true`, 那么 i 就是素数。如图 22-3 所示, 2、3、5、7、11、13、17、19、23 都是素数。程序清单 22-7 给出使用埃拉托色尼筛选算法求素数的程序。

程序清单 22-7 SieveOfEratosthenes.java

```

1 import java.util.Scanner;
2
3 public class SieveOfEratosthenes {
4     public static void main(String[] args) {
5         Scanner input = new Scanner(System.in);
6         System.out.print("Find all prime numbers <= n, enter n: ");
7         int n = input.nextInt();
8
9         boolean[] primes = new boolean[n + 1]; // Prime number sieve
10
11         // Initialize primes[i] to true
12         for (int i = 0; i < primes.length; i++) {
13             primes[i] = true;
14         }
15
16         for (int k = 2; k <= n / k; k++) {
17             if (primes[k]) {
18                 for (int i = k; i <= n / k; i++) {
19                     primes[k * i] = false; // k * i is not prime
20                 }
21             }
22         }
23     }
24 }
```

```

23
24     int count = 0; // Count the number of prime numbers found so far
25     // Print prime numbers
26     for (int i = 2; i < primes.length; i++) {
27         if (primes[i]) {
28             count++;
29             if (count % 10 == 0)
30                 System.out.printf("%7d\n", i);
31             else
32                 System.out.printf("%7d", i);
33         }
34     }
35
36     System.out.println("\n" + count +
37         " prime(s) less than or equal to " + n);
38 }
39 }

```

```

Find all prime numbers <= n, enter n: 1000 
The prime numbers are:
    2      3      5      7      11      13      17      19      23      29
   31     37     41     43     47     53     59     61     67     71
...
...
168 prime(s) less than or equal to 1000

```

注意， $k \leq n/k$ (第 16 行)。否则， $k*i$ 可能会大于 n (第 19 行)。该算法的时间复杂度是多少？

对于每个素数 k (第 17 行)，算法将 $\text{primes}[k*i]$ 设置为 `false` (第 19 行)。这在 `for` 循环中执行了 $n/k - k + 1$ 次 (第 18 行)。这样，找出不超过 n 的所有素数的复杂度就是

$$\begin{aligned}
 & \frac{n}{2} - 2 + 1 + \frac{n}{3} - 3 + 1 + \frac{n}{5} - 5 + 1 + \frac{n}{7} - 7 + 1 + \frac{n}{11} - 11 + 1 \cdots \\
 &= O\left(\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \frac{n}{11} + \cdots\right) < O(n\pi(n)) \\
 &= O\left(n \frac{\sqrt{n}}{\log n}\right) \quad \swarrow \text{该数列的项数为 } \pi(n)
 \end{aligned}$$

该上限 $O\left(\frac{n\sqrt{n}}{\log n}\right)$ 是非常松散的。实际的时间复杂度比 $O\left(\frac{n\sqrt{n}}{\log n}\right)$ 好很多。埃拉托色尼筛选法对于小的 n 值而言是一个好的算法，这样 `primes` 数组可以载入内存。

表 22-5 总结了找出不超过 n 的所有素数的三个算法的复杂度。

表 22-5 素数算法的比较

算法	复杂度	描述
程序清单 5-15	$O(n^2)$	穷举法，检测所有可能的因子
程序清单 22-5	$O(n \sqrt{n})$	检测直到 \sqrt{n} 的因子
程序清单 22-6	$O\left(\frac{n\sqrt{n}}{\log n}\right)$	检测直到 \sqrt{n} 的素数因子
程序清单 22-7	$O\left(\frac{n\sqrt{n}}{\log n}\right)$	埃拉托色尼筛选算法

复习题

- 22.17 证明如果 n 不是素数, 那么必然存在一个素数 p , 使得 $P \leq \sqrt{n}$ 并且 p 是 n 的一个因子。
- 22.18 描述埃拉托色尼筛选算法是如何找到素数的。

22.8 使用分而治之法寻找最近的点对

要点提示: 本节给出使用分而治之法找到最近点对的高效算法。

给定一个点集, 那么最近的点对问题就是找出两个距离最近的点。如图 22-4 所示, 在最近点对动画中画出一条直线连接最近的两个点。

8.6 节给出了一个求最近点对的穷举算法。该算法计算所有点对之间的距离, 并且求出最小距离的点对。显然, 这个算法耗费 $O(n^2)$ 时间。可以设计一个更有效的算法吗?

我们将使用一种称为分而治之 (divide-and-conquer) 的方法来解决这个问题。该方法将问题分解为子问题, 解决子问题, 然后将子问题的解答合并从而获得整个问题的解答。和动态编程方法不一样的是, 分而治之方法中的子问题不会交叉。子问题类似初始问题, 但是具有更小的尺寸, 因此可以应用递归来解决这样的问题。事实上, 所有递归问题的解答都遵循分而治之方法。

程序清单 22-8 描述了如何使用分而治之的方法来解决最近点对问题。

程序清单 22-8 寻找最近点对的算法

步骤 1: 以 x 坐标的升序对点进行排序。对于 x 坐标一样的点, 按它的 y 坐标排序。这样就能得到一个由排好序的点构成的线性表 S 。

步骤 2: 使用排好序的线性表的中点将 S 分为两个大小相等的子集 S_1 和 S_2 。让中点位于 S_1 中。递归地找到 S_1 和 S_2 中的最近点对。设 d_1 和 d_2 分别表示两个子集中最近点对的距离。

步骤 3: 找出 S_1 中的点和 S_2 中的点之间距离最近的点对, 它们之间的距离用 d_3 表示。最近的点对是距离为 $\min(d_1, d_2, d_3)$ 的点对。

选择排序耗费 $O(n^2)$ 时间。在第 23 章, 我们将介绍归并排序和堆排序。这些排序算法耗费 $O(n \log n)$ 时间。所以, 步骤 1 可以在 $O(n \log n)$ 时间内完成。

步骤 3 可以在 $O(n)$ 时间内完成。设 $d = \min(d_1, d_2)$ 。我们已经了解到最近点对的距离不可能大于 d 。对于 S_1 中的点和 S_2 中的点, 形成一个最近点对集 S , 左边的点必须在 stripL 中, 而右边的点必须在 stripR 中, 如图 22-5a 所示。

对于 stripL 中的点 P , 只需要考虑在 $d \times 2d$ 的矩形中的右点, 如图 22-5b 所示。矩形外的任何右点都不能与 P 形成最近点对。因为在 S_2 中最近点对的距离大于或等于 d , 在矩形中最多有 6 个点。因此, 对于 stripL 中的每个点, 最多考虑 stripR 中的 6 个点。

对于 stripL 中的每个点 p , 如何定位在 stripR 对应的 $d \times 2d$ 的矩形中的点? 如果 stripL 和 stripR 的点都以 y 坐标的升序排列, 这是可以很高效地完成的。设 pointsOrderedOnY 是一个以 y 坐标升序排列的点构成的线性表, 它可以在算法中提前获得。 stripL 和 stripR 都可以从步骤 3 步的 pointsOrderedOnY 中获取, 如程序清单 22-9 所示。

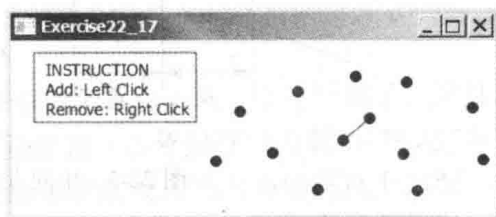


图 22-4 最近点对动画中, 当交互式地增加和移除点时, 画一条直线动态连接最近的点对

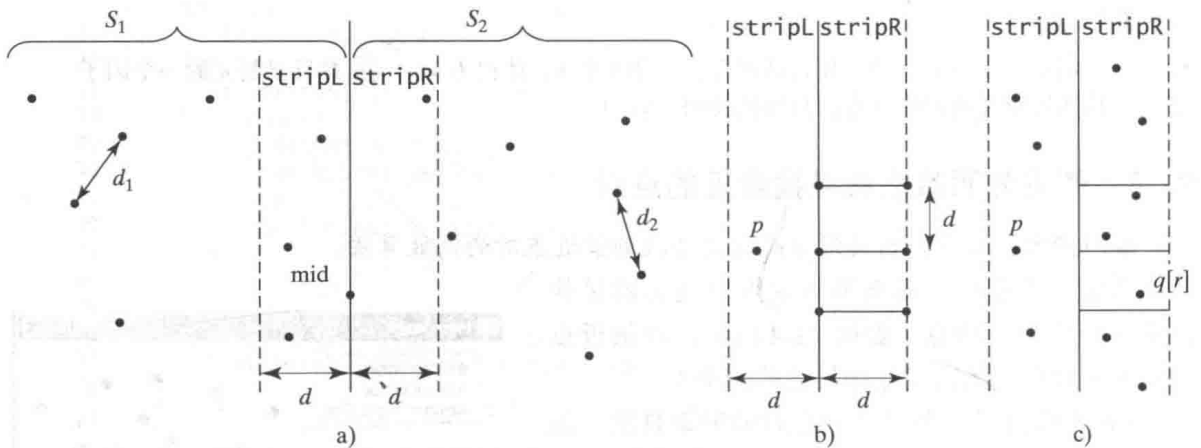


图 22-5 中间点将点集分为两个相同大小的集合

程序清单 22-9 获取 stripL 和 stripR 的算法

```

1 for each point p in pointsOrderedOnY
2   if (p is in S1 and mid.x - p.x <= d)
3     append p to stripL;
4   else if (p is in S2 and p.x - mid.x <= d)
5     append p to stripR;

```

假设 stripL 中的点和 stripR 中的点分别是 $\{p_0, p_1, \dots, p_k\}$ 和 $\{q_0, q_1, \dots, q_l\}$, 如图 22-5c 所示。stripL 中的点和 stripR 中的点之间的最近点对可以使用程序清单 22-10 所描述的算法求得。

程序清单 22-10 在步骤 3 找出最近点对的算法

```

1 d = min(d1, d2);
2 r = 0; // r is the index of a point in stripR
3 for (each point p in stripL) {
4   // Skip the points in stripR below p.y - d
5   while (r < stripR.length && q[r].y <= p.y - d)
6     r++;
7
8   let r1 = r;
9   while (r1 < stripR.length && |q[r1].y - p.y| <= d) {
10    // Check if (p, q[r1]) is a possible closest pair
11    if (distance(p, q[r1]) < d) {
12      d = distance(p, q[r1]);
13      (p, q[r1]) is now the current closest pair;
14    }
15
16    r1 = r1 + 1;
17  }
18 }

```

以 p_0, p_1, \dots, p_k 的顺序考虑 stripL 中的点。对于 stripL 中的点 p , 跳过 stripR 中在 $p.y-d$ 下面的点 (第 5~6 行)。一旦跳过某个点, 这个点就不再考虑。while 循环 (第 9~17 行) 检测 $(p, q[r1])$ 是否是可能的最近点对。这里最多有 6 个这样的 $q[r1]$, 因为 stripR 中的两点距离不能小于 d 。因此在步骤 3 找出最近点对的复杂度是 $O(n)$ 。

注意: 程序清单 22-8 中的步骤 1 只被执行一次以预先对点进行排序。假设所有的点都预先排好序了。设 $T(n)$ 表示算法的复杂度, 那么

$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

因此,找出最近点对耗费的时间是 $O(n\log n)$ 。这个算法的完整实现留作练习题(参见编程练习题 22.7)。

复习题

22.19 什么是分而治之方法? 给出一个示例。

22.20 分而治之以及动态编程之间的区别是什么?

22.21 可以使用分而治之法设计一个算法以找到一个线性表中的最小元素吗? 这个算法的复杂度是多少?

22.9 使用回溯法解决八皇后问题

要点提示: 本节使用回溯法解决八皇后问题。

八皇后问题是要找到一个解决方案,可以在一个棋盘的每行上放一个皇后棋子,并且没有两个皇后可以相互攻击。这个问题可以用递归方法解决(参见编程练习题 18.34)。本节中,我们将介绍一个称为回溯法(backtracking)的通用算法设计技术来解决这个问题。回溯法渐进地寻找一个备选方案,一旦确定该备选方案不可能是一个有效方案的时候则放弃掉,继而寻找一个新的备选方案。

可以使用一个二维数组来表示一个棋盘。然而,由于每行只能放一个皇后,因此使用一个一维数组足以表示每行皇后的位置了。可以如下定义一个 `queens` 数组:

```
int[] queens = new int[8];
```

将 `queens[i]` 赋值为 `j` 表示一个皇后放置在第 `i` 行第 `j` 列。图 22-6a 显示了表示图 22-6b 中棋盘的 `queens` 数组的内容。

搜索从 $k=0$ 的第一行开始,其中 k 为考察的当前行的下标。算法为当前行检测一个皇后是否可能放在第 j 列,按照 $j=0,1,\dots,7$ 的次序。搜索以下列步骤进行:

- 如果成功了,则继续为下一行的皇后搜索一个位置。如果当前行是最后一行,则解决方案已经找到。
- 如果不成功,则回溯到前一行,继续在前一行的下一列上搜索一个新的放置位置。
- 如果算法回溯到第一行并且不能在该行为一个皇后找到一个新的位置,则不能找到方案。

算法的运行过程动画可以参见网址 www.cs.armstrong.edu/liang/animation/EightQueens-Animation.html。

程序清单 22-11 给出了显示八皇后问题解决方案的程序。

程序清单 22-11 EightQueens.java

```
1 import javafx.application.Application;
2 import javafx.geometry.Pos;
3 import javafx.stage.Stage;
4 import javafx.scene.Scene;
5 import javafx.scene.control.Label;
6 import javafx.scene.image.Image;
7 import javafx.scene.image.ImageView;
8 import javafx.scene.layout.GridPane;
```

queens[0]	0
queens[1]	4
queens[2]	7
queens[3]	5
queens[4]	2
queens[5]	6
queens[6]	1
queens[7]	3

a)
b)



图 22-6 `queens[i]` 表示第 i 行的皇后的位置

```

9
10 public class EightQueens extends Application {
11     public static final int SIZE = 8; // The size of the chess board
12     // queens are placed at (i, queens[i])
13     // -1 indicates that no queen is currently placed in the ith row
14     // Initially, place a queen at (0, 0) in the 0th row
15     private int[] queens = {-1, -1, -1, -1, -1, -1, -1, -1};
16
17     @Override // Override the start method in the Application class
18     public void start(Stage primaryStage) {
19         search(); // Search for a solution
20
21         // Display chess board
22         GridPane chessBoard = new GridPane();
23         chessBoard.setAlignment(Pos.CENTER);
24         Label[][] labels = new Label[SIZE][SIZE];
25         for (int i = 0; i < SIZE; i++)
26             for (int j = 0; j < SIZE; j++) {
27                 chessBoard.add(labels[i][j] = new Label(), j, i);
28                 labels[i][j].setStyle("-fx-border-color: black");
29                 labels[i][j].setPrefSize(55, 55);
30             }
31
32         // Display queens
33         Image image = new Image("image/queen.jpg");
34         for (int i = 0; i < SIZE; i++)
35             labels[i][queens[i]].setGraphic(new ImageView(image));
36
37         // Create a scene and place it in the stage
38         Scene scene = new Scene(chessBoard, 55 * SIZE, 55 * SIZE);
39         primaryStage.setTitle("EightQueens"); // Set the stage title
40         primaryStage.setScene(scene); // Place the scene in the stage
41         primaryStage.show(); // Display the stage
42     }
43
44     /** Search for a solution */
45     private boolean search() {
46         // k - 1 indicates the number of queens placed so far
47         // We are looking for a position in the kth row to place a queen
48         int k = 0;
49         while (k >= 0 && k < SIZE) {
50             // Find a position to place a queen in the kth row
51             int j = findPosition(k);
52             if (j < 0) {
53                 queens[k] = -1;
54                 k--; // back track to the previous row
55             } else {
56                 queens[k] = j;
57                 k++;
58             }
59         }
60
61         if (k == -1)
62             return false; // No solution
63         else
64             return true; // A solution is found
65     }
66
67     public int findPosition(int k) {
68         int start = queens[k] + 1; // Search for a new placement
69
70         for (int j = start; j < SIZE; j++) {
71             if (isValid(k, j))
72                 return j; // (k, j) is the place to put the queen now

```

```

73     }
74
75     return -1;
76 }
77
78 /** Return true if a queen can be placed at (row, column) */
79 public boolean isValid(int row, int column) {
80     for (int i = 1; i <= row; i++)
81         if (queens[row - i] == column // Check column
82             || queens[row - i] == column - i // Check upleft diagonal
83             || queens[row - i] == column + i) // Check upright diagonal
84             return false; // There is a conflict
85     return true; // No conflict
86 }
87 }

```

程序调用 `search()` (第 19 行) 来搜索一个解决方案。开始时, 任何一行上都没有皇后 (第 15 行)。现在搜索从 $k = 0$ 的第一行开始 (第 48 行), 找到一个位置放置皇后 (第 51 行)。如果成功了, 则将其放置在该行上 (第 56 行) 然后考虑下一行 (第 57 行)。如果没有成功, 则回溯到前一行 (第 53 ~ 54 行)。

`findPosition(k)` 方法为在第 k 行放置一个皇后寻找一个可能的位置, 从 `queen[k] + 1` 开始搜索 (第 68 行)。它依次检测一个皇后是否可以放置在 `start, start + 1, …`, 直到 7 (第 70 ~ 73 行)。如果可以, 则返回列的下标 (第 72 行); 否则, 返回 `-1` (第 75 行)。

`isValid(row, column)` 方法用于检测放置一个皇后在指定位置是否会引起和之前所放置的皇后的冲突 (第 71 行)。它确保皇后没有被放置在同一列上 (第 81 行)、左上角对角线上 (第 82 行) 或者右上角对角线上 (第 83 行), 如图 22-7 所示。

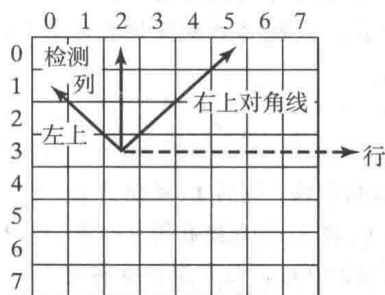


图 22-7 调用 `isValid(row, column)` 检测一个皇后是否可以放置在 `(row, column)`

✓ 复习题

22.22 什么是回溯? 给出一个示例。

22.23 如果将八皇后问题推广到在 $n \times n$ 棋盘上的 n 皇后问题, 算法的复杂度将是多少?

22.10 计算几何: 寻找凸包

🔑 要点提示: 本节给出在点集中寻找凸包的一个高效算法。

计算几何为几何问题研究算法。它在计算机图形学、游戏、模式识别、图像处理、机器人、地理信息系统以及计算机辅助设计和制造方面有着广泛应用。22.8 节给出了一个寻找最近点对的几何算法。本节介绍一个寻找凸包的几何算法。

给定一个点集, 凸包 (convex hull) 是指包围所有这些点的最小凸多边形, 如图 22-8a 所示。如果连接两个顶点的任意直线都在多边形里面, 则这个多边形是凸的。例如, 图

22-8a 中的顶点 v_0, v_1, v_2, v_3, v_4 以及 v_5 构成了一个凸多边形, 但是图 22-8b 中的不是, 因为连接 v_3 和 v_1 的点不在多边形里面。

凸包在游戏编程、模式识别和图像处理方面有许多应用。在介绍算法之前, 使用网址 www.cs.armstrong.edu/liang/animation/ConvexHull.html 的交互式工具来熟悉概念是有帮助的, 如图 22-8c 所示。通过这个工具可以添加和移除一些点, 然后动态地显示相应的凸包。

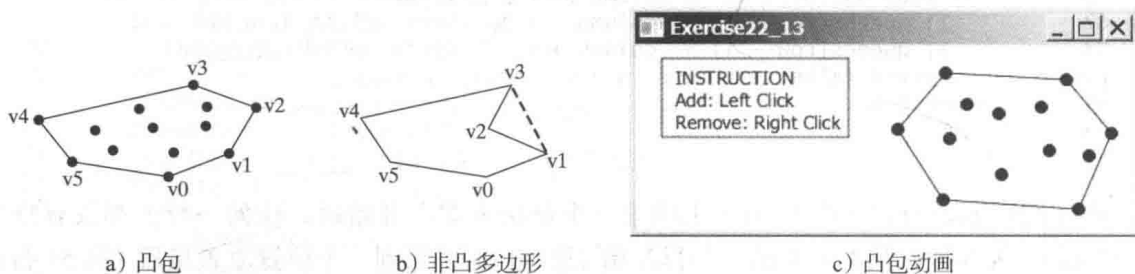


图 22-8 一个凸包是包含一个点集的最小凸多边形

已经有许多用于寻找一个凸包的算法, 本节介绍两个流行的算法: 卷包裹算法和格雷厄姆算法。

22.10.1 卷包裹算法

卷包裹算法 (gift-wrapping algorithm) 是一种直观方法, 工作机制如程序清单 22-12 所示。

程序清单 22-12 使用卷包裹算法寻找凸包

步骤 1: 给定一个点的线性表 S , 将 S 中的点标记为 s_0, s_1, \dots, s_k 。选择 S 中最右下角的点, 如图 22-9a 所示, h_0 即为这样的点, 将 h_0 添加到线性表 H 中 (H 初始时空, 当算法结束时, H 将容纳凸包中的所有点), 将 t_0 赋为 h_0 。

步骤 2: 将 t_1 赋值为 s_0 。

对于 S 中的每个点 p ,

如果 p 在从 t_0 到 t_1 的连接直线的右侧, 则将 t_1 赋值为 p 。

(步骤 2 之后, 没有点存在于从 t_0 到 t_1 的直线右侧, 如图 22-9b 所示。)

步骤 3: 如果 t_1 为 h_0 (参见图 22-9d), 则 H 中的点构建了一个 S 点集的凸包。否则, 将 t_1 添加到 H 中, 将 t_0 赋值为 t_1 , 回到步骤 2 (参见图 22-9c)。

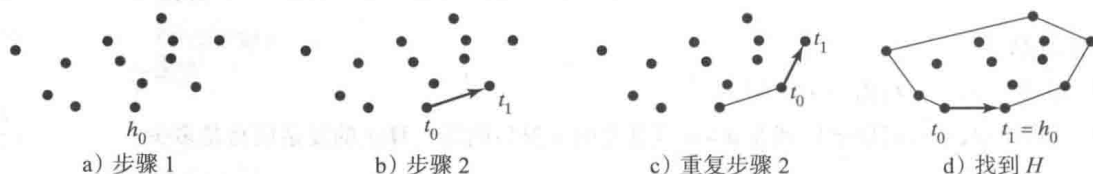


图 22-9 a) h_0 为 S 中最右下角的点; b) 步骤 2 找到点 t_1 ; c) 凸包不断重复扩张; d) 当 t_1 成为 h_0 时, 一个凸包被找到

凸包渐进地扩张。正确性由以下事实确保: 步骤 2 后, 没有点存在于从 t_0 到 t_1 的连线的右侧。这保证了连接 S 中两个点的每条线段都位于多边形里面。

步骤 1 中寻找最右下方的点可以在 $O(n)$ 时间内完成。无论点在直线的左侧、右侧, 还是在直线上, 可以在 $O(1)$ 时间内确定 (参见编程练习题 3.32)。因此, 步骤 2 中将耗费 $O(n)$ 时间找到一个新的点 t_1 。步骤 2 重复 h 次, 其中 h 为凸包的边数。因此, 算法耗费 $O(hn)$ 时

间。最坏情况下， h 等于 n 。

该算法的实现留作练习题（参见编程练习题 22.9）。

22.10.2 格雷厄姆算法

一种更为有效的算法是 1972 年由罗纳德·格雷厄姆（Ronald Graham）开发的，如程序清单 22-13 所示。

程序清单 22-13 使用格雷厄姆算法找到凸包

步骤 1：给定一个点的线性表 S ，选择 S 中最右下角的点 p_0 ，如图 22-10a 所示， p_0 即为这样的点。

步骤 2：将 S 中的点按照以 p_0 为原点的 x 轴夹角进行排序，如图 22-10b 所示。如果出现同样的值，即两个点具有同样的角度，则弃掉离 p_0 较近的那个点。 S 中的点现在排序为 $p_0, p_1, p_2, \dots, p_{n-1}$ 。

步骤 3：将 p_0, p_1 和 p_2 加入栈 H （算法结束后， H 包含凸包中的所有点）。

步骤 4：

```
i = 3;
while ( i < n ){
    让  $t_1$  和  $t_2$  作为栈  $H$  中顶部的第 1 个和第 2 个元素;
    if (  $p_i$  位于  $t_2$  到  $t_1$  的连线的左侧 ) {
        将  $p_i$  压入栈  $H$ ;
        i ++; // 考察  $S$  中的下一个点
    }
    else
        将栈  $H$  的顶部元素弹出
}
```

步骤 5： H 中的点形成了一个凸包。

凸包是逐步找到的。首先， p_0, p_1 以及 p_2 构成了一个凸包。 p_3, p_3 在当前的凸包之外，因为点是根据它们的角度按照增序进行排列的。如果 p_3 严格地位于从 p_1 到 p_2 的连线的左侧（参见图 22-10c），则将 p_3 压入 H 中。现在 p_0, p_1, p_2 以及 p_3 构建了一个凸包。如果 p_3 在从 p_1 到 p_2 的连线的右侧（参见图 22-10d），则将 p_2 从 H 中弹出，并且将 p_3 压入 H 中。现在 p_0, p_1, p_3 形成了一个凸包，而 p_2 位于凸包中间。可以通过推理证明，步骤 5 H 中所有点针对输入点列表 S 的点构建了一个凸包。

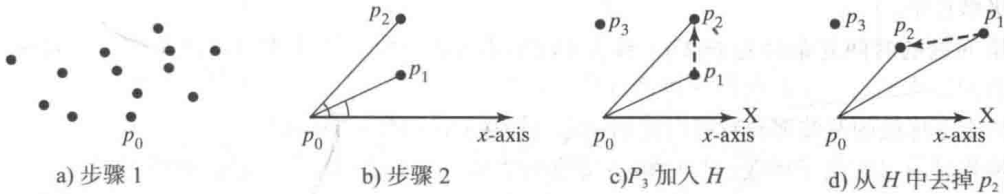


图 22-10 a) p_0 是 S 中最右边下角的点；b) 点根据它们的夹角排序；c) ~ d) 凸包渐进被找到

步骤 1 中寻找最右下角的点可以在 $O(n)$ 时间内找到。角度可以使用三角函数进行计算。然而，可以不计算角度而进行排序。观察到当且仅当 p_2 位于从 p_0 到 p_1 的连线左侧时， p_2 将比 p_1 构建一个更大的角度。一个点是否位于一条直线的左侧可以在 $O(1)$ 时间内确定，如编程练习题 3.32 所示。步骤 2 的排序可以使用归并排序或者堆排序算法在 $O(n \log n)$ 时间内完成，这两种排序算法将在第 23 章中介绍。步骤 4 可以在 $O(n)$ 时间内完成。因此，算法需要

$O(n \log n)$ 时间。

该算法的实现留作练习题 (参见编程练习题 22.11)。

复习题

22.24 什么是凸包?

22.25 描述如何使用卷包裹算法来找到凸包。列表 H 需要使用 ArrayList 或者 LinkedList 来实现吗?

22.26 描述如何使用格雷厄姆算法来找到凸包。为什么算法使用栈来存储凸包中的点?

关键术语

average-case analysis (平均情况分析)

backtracking approach (回溯法)

best-case input (最佳情况输入)

big O notation (大 O 标记)

brute force (穷举法)

constant time (常量时间)

convex hull (凸包)

divide-and-conquer approach (分而治之法)

dynamic programming approach (动态编程法)

exponential time (指数时间)

growth rate (增长率)

logarithmic time (对数时间)

quadratic time (二次时间)

space complexity (空间复杂度)

time complexity (时间复杂度)

worst-case input (最差情况输入)

本章小结

1. 大 O 标记是分析算法性能的理论方法。它估计算法的执行时间随着输入规模的增加会有多快的增长。因此, 可以通过检查两个算法的增长率来比较它们。
2. 导致最短执行时间的输入称为最佳情况输入, 而导致最长执行时间的输入称为最差情况输入。最佳情况和最差情况都不具有代表性, 但是最差情况分析非常有用。你可以确保算法永远不会比最差情况还慢。
3. 平均情况分析试图在所有可能的相同规模的输入中确定平均时间。平均情况分析是比较理想的, 但是完成很困难, 因为对于许多问题, 要确定不同输入实例的相对概率和分布是相当困难的。
4. 如果执行时间与输入规模无关, 我们就说该算法耗费了常量时间, 以符号 $O(1)$ 表示。
5. 线性查找耗费 $O(n)$ 时间。具有 $O(n)$ 时间复杂度的算法称为线性算法, 它表现为线性增长率。二分查找耗费 $O(\log n)$ 时间。具有 $O(\log n)$ 时间复杂度的算法称为对数算法, 它表现为对数增长率。
6. 选择排序的最差情况时间复杂度为 $O(n^2)$ 。具有 $O(n^2)$ 时间复杂度的算法称为平方级算法, 它表现为平方级增长率。
7. 汉诺塔问题的时间复杂度是 $O(2^n)$ 。具有 $O(2^n)$ 时间复杂度的算法称为指数算法, 它表现为指数增长率。
8. 求出给定下标处的斐波那契数可以使用动态编程在 $O(n)$ 时间内求解。
9. 动态编程是通过解决子问题, 然后将子问题的结果结合起来获得整个问题的解的过程。动态编程的关键思想是只解决子问题一次, 并将子问题的结果存储以备后用, 从而避免了重复的子问题的求解。
10. 欧几里得的 GCD 算法需要 $O(\log n)$ 时间。
11. 所有小于等于 n 的素数可以在 $O\left(\frac{n\sqrt{n}}{\log n}\right)$ 时间内找到。
12. 使用分而治之法可以在 $O(n \log n)$ 时间内找到最近点对。
13. 分而治之法将问题分解为子问题, 解决子问题, 然后将子问题的解答合并从而获得整个问题的解答。和动态编程不一样的是, 分而治之法中的子问题不会交叉。子问题类似初始问题, 但是具有更小的尺寸, 因此可以应用递归来解决这样的问题。

- 14. 可以使用回溯法解决八皇后问题。
- 15. 回溯法渐进地寻找一个备选方案，一旦确定该备选方案不可能是一个有效方案，则放弃掉，继而寻找一个新的备选方案。
- 16. 使用卷包裹法可以在 $O(n^2)$ 时间内找到一个点集的凸包，使用格雷厄姆算法则需要 $O(n\log n)$ 时间。

测试题

回答位于网址 www.cs.armstrong.edu/liang/intro10e/quiz.html 的本章测试题。

编程练习题

*22.1 (最大连续递增的有序子串) 编写一个程序，提示用户输入一个字符串，然后显示最大连续递增的有序子串。分析你的程序的时间复杂度。下面是一个运行示例：

```
Enter a string: abcabcdgabxy
abcdg

Enter a string: abcabcdgabmnsxy
abmnsxy
```

**22.2 (最大增序子序列) 编写一个程序，提示用户输入一个字符串，然后显示最大的增序子串。分析你的程序的时间复杂度。下面是一个运行示例：

```
Enter a string: Welcome
Welo
```

*22.3 (模式匹配) 编写一个程序，提示用户输入两个字符串，然后检测第二个字符串是否是第一个字符串的子串。假定在字符串中相邻的字符是不同的。(不要使用 String 类中的 indexOf 方法。) 分析你的算法的时间复杂度。你的算法至少需要 $O(n)$ 时间。下面是该程序的一个运行示例：

```
Enter a string s1: Welcome to Java
Enter a string s2: come
matched at index 3
```

*22.4 (模式匹配) 编写一个程序，提示用户输入两个字符串，然后检测第二个字符串是否是第一个字符串的子串。(不要使用 String 类中的 indexOf 方法。) 分析你的算法的时间复杂度。下面是该程序的一个运行示例：

```
Enter a string s1: Mississippi
Enter a string s2: sip
matched at index 6
```

*22.5 (同样个数的子序列) 编写一个程序，提示用户输入一个以 0 结束的整数序列，找出有同样数字的最长的子序列。下面是该程序的一个运行示例：

```
Enter a series of numbers ending with 0:
2 4 4 8 8 8 8 2 4 4 0
The longest same number sequence starts at index 3 with 4 values of 8
```

*22.6 (GCD 的执行时间) 编写一个程序，使用程序清单 22-3 和程序清单 22-4 中的算法，求下标从 40 到 45 的每两个连续的斐波那契数的 GCD，获取其执行时间。你的程序应该打印如下所示的一个表格：

	40	41	42	43	44	45
程序清单 22.3 GCD						
程序清单 22.4 GCDEuclid						

(提示：可以使用下面的代码模板来获取执行时间。)

```
long startTime = System.currentTimeMillis();
perform the task;
long endTime = System.currentTimeMillis();
long executionTime = endTime - startTime;
```

****22.7** (最近的点对) 22.8 节介绍了一个使用分而治之方法求最近点对的算法。实现这个算法，使其满足下面的要求：

- 使用和编程练习题 20.4 相同的方式定义类 `Point` 和 `CompareY`。
- 定义一个名为 `Pair` 的类，它的数据域 `p1` 和 `p2` 表示两个点，名为 `getDistance()` 的方法返回这两个点之间的距离。
- 实现下面的方法：

```
/** Return the distance of the closest pair of points */
public static Pair getClosestPair(double[][] points)

/** Return the distance of the closest pair of points */
public static Pair getClosestPair(Point[] points)

/** Return the distance of the closest pair of points
 * in pointsOrderedOnX[low..high]. This is a recursive
 * method. pointsOrderedOnX and pointsOrderedOnY are
 * not changed in the subsequent recursive calls.
 */
public static Pair distance(Point[] pointsOrderedOnX,
    int low, int high, Point[] pointsOrderedOnY)

/** Compute the distance between two points p1 and p2 */
public static double distance(Point p1, Point p2)

/** Compute the distance between points (x1, y1) and (x2, y2) */
public static double distance(double x1, double y1,
    double x2, double y2)
```

****22.8** (不大于 10 000 000 000 的所有素数) 编写一个程序，找出不大于 10 000 000 000 的所有素数。大概有 455 052 511 个这样的素数。你的程序应该满足下面的要求：

- 应该将这些素数都存储在一个名为 `PrimeNumber.dat` 的二进制数据文件中。当找到一个新素数时，将该数字追加到这个文件中。
- 为了判定一个新数是否是素数，程序应该从数据文件加载这些素数到一个大小为 10 000 的 `long` 型的数组中。如果数组中没有任何数是这个新数的除数，继续从该数据文件中读取下 10 000 个素数，直到找到除数或者读取完文件中的所有数字。如果没找到除数，这个新的数字就是素数。
- 因为执行该程序要花很长时间，所以应该把它作为 UNIX 机器上的一个批处理任务来运行。如果机器被关闭或重启，程序应该使用二进制数据文件中存储的素数来继续，而不是从零开始启动。

****22.9** (几何：找到凸包的卷包裹算法) 22.10.1 节介绍了为一个点集找到一个凸包的卷包裹算法。假定使用 Java 的坐标系表示点，使用下面的方法实现该算法：

```
/** Return the points that form a convex hull */
public static ArrayList<Point2D> getConvexHull(double[][] s)

Point2D 在 9.6 节中定义。
```

编写一个测试程序，提示用户输入点集的大小以及点，然后显示构成一个凸包的点的信息。下面是一个运行示例：

```
How many points are in the set? 6 Enter
Enter 6 points: 1 2.4 2.5 2 1.5 34.5 5.5 6 6 2.4 5.5 9 Enter
The convex hull is
(1.5, 34.5) (5.5, 9.0) (6.0, 2.4) (2.5, 2.0) (1.0, 2.4)
```

- 22.10 (素数的个数) 编程练习题 22.8 将素数存储在一个名为 PrimeNumbers.dat 的文件中。编写一个程序，找出小于或等于 10、100、1 000、10 000、100 000、1 000 000、10 000 000、100 000 000、1 000 000 000、10 000 000 000 的素数个数。你的程序应该从 PrimeNumbers.dat 文件中读取数据。
- **22.11** (几何：寻找凸包的格雷厄姆算法) 22.10.2 节介绍了为一个点集寻找凸包的格雷厄姆算法。假定使用 Java 的坐标系统表示点。使用下面的方法实现该算法：

```
/** Return the points that form a convex hull */
public static ArrayList<MyPoint> getConvexHull(double[][] s)
```

MyPoint is a static inner class defined as follows:

```
private static class MyPoint implements Comparable<MyPoint> {
    double x, y;

    MyPoint rightMostLowestPoint;

    MyPoint(double x, double y) {
        this.x = x; this.y = y;
    }

    public void setRightMostLowestPoint(MyPoint p) {
        rightMostLowestPoint = p;
    }

    @Override
    public int compareTo(MyPoint o) {
        // Implement it to compare this point with point o
        // angularly along the x-axis with rightMostLowestPoint
        // as the center, as shown in Figure 22.10b. By implementing
        // the Comparable interface, you can use the Array.sort
        // method to sort the points to simplify coding.
    }
}
```

编写一个测试程序，提示用户输入点集的大小和点，然后显示构成一个凸包的点。下面是一个运行示例：

```
How many points are in the set? 6
Enter 6 points: 1 2.4 2.5 2 1.5 34.5 5.5 6 6 2.4 5.5 9
The convex hull is
(1.5, 34.5) (5.5, 9.0) (6.0, 2.4) (2.5, 2.0) (1.0, 2.4)
```

- *22.12 (最后的 100 个素数) 编程练习题 22.8 将素数存储在一个名为 PrimeNumbers.dat 的文件中。编写一个高效程序，从该文件中读取最后 100 个素数。
(提示：不要从文件中读取所有的数字，跳过文件中最后 100 个数之前的所有数。)
- **22.13** (几何：凸包动画) 编程练习题 22.11 为从控制台输入的点集中找到凸包。编写一个程序，可以让用户通过单击鼠标左 / 右键来添加 / 移除点，然后显示凸包，如图 22-8c 所示。
- *22.14 (素数的执行时间) 编写一个程序，使用程序清单 22-5 ~ 程序清单 22-7 中的算法，找出小于 8 000 000、10 000 000、12 000 000、14 000 000、16 000 000 和 18 000 000 的所有素数，获取其执行时间。你的程序应该打印如下所示的一个表格：

	8000000	10000000	12000000	14000000	16000000	18000000
程序清单 22.5						
程序清单 22.6						
程序清单 22.7						

- **22.15** (几何：无交叉多边形) 编写一个程序，可以让用户通过单击鼠标左 / 右键来添加 / 移除点，然

后显示一个连接所有点的无交叉多边形, 如图 22-11a 所示。如果一个多边形有两条或者更多的边是相交的, 则认为是交叉多边形, 如图 22-11b 所示。使用如下算法来从一个点集中构建一个多边形。

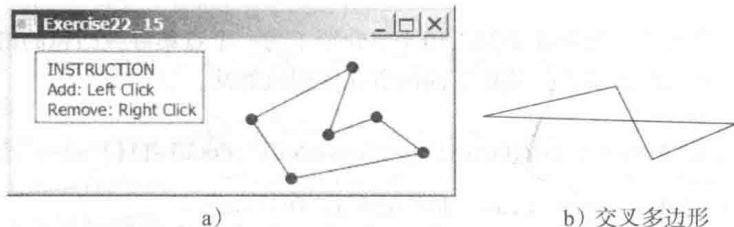


图 22-11 a) 编程练习题 22.15 为一个点集显示一个无交叉多边形; b) 在一个交叉多边形中, 两条或者更多的边是相交的

步骤 1: 给定一个点集 S , 选择 S 中的最右下角点 p_0 。

步骤 2: 将 S 中的点按照以 p_0 为原点的 x 轴夹角进行排序。如果出现同样的值, 即两个点具有同样的角度, 则认为离 p_0 较近的那个点具有更大的角度。 S 中的点现在排序为 $p_0, p_1, p_2, \dots, p_{n-1}$ 。

步骤 3: 排好序的点形成了一个无交叉多边形。

- **22.16 (线性查找动画)** 编写一个程序, 显示线性查找的动画。创建一个包含从 1 到 20 的 20 个不同数字并且顺序随机的数组。数组元素以直方图显示, 如图 22-12 所示。你需要在文本域中输入一个查找键值。单击 **step** 按钮将引发程序执行算法中的一次比较, 重绘直方图并且其中一个条形显示查找的位置。这个按钮同时冻结文本域以防止其中的值被改变。当算法结束时, 在 **border** 面板的顶部标签中显示状态, 从而给出用户信息。单击 **Reset** 按钮创建一个新的随机数组, 从而开始一次新的查找。这个按钮也使得文本域可以编辑。

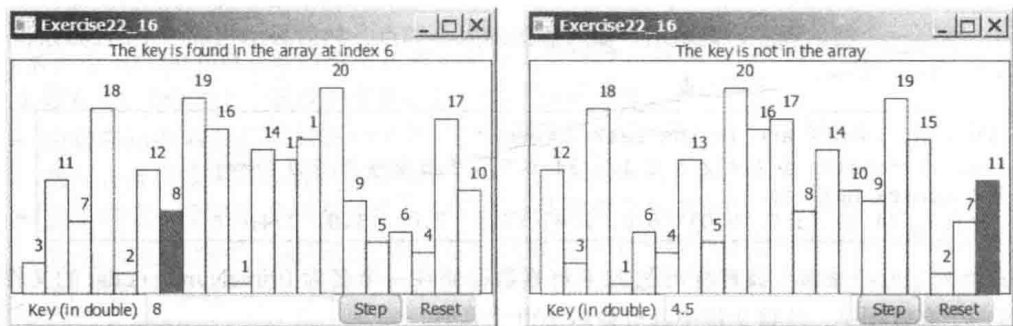


图 22-12 程序显示线性查找的动画

- **22.17 (最近点对的动画)** 编写一个程序, 可以让用户通过单击鼠标左 / 右键来添加 / 移除点, 然后显示一条连接最近点对的直线, 如图 22-4 所示。
- **22.18 (二分查找动画)** 编写一个程序, 显示二分查找的动画。创建一个包含从 1 到 20 的顺序数字的数组。数组元素以直方图显示, 如图 22-13 所示。你需要在文本域中输入一个搜索键值。单击 **Step** 按钮将引发程序执行算法中的一次比较。使用淡灰色来绘制代表目前查找范围内的数字的条形, 使用黑色绘制表示查找范围的中间数的条形。**Step** 按钮同时冻结文本域以防止其中的值被改变。当算法结束时, 在 **border** 面板的顶部标签中显示状态信息。单击 **Reset** 按钮创建一个新的随机数组, 从而开始一次新的查找。这个按钮也使得文本域可以编辑。
- *22.19 (最大块)** 编程练习题 8.35 描述了寻找最大块的问题。设计一个动态编程的算法, 从而在 $O(n^2)$ 时间内求解这个问题。编写一个测试程序, 显示一个 10×10 的方格矩阵, 如图 22-14a 所示。

矩阵中的每个元素为 0 或者 1，单击 Refresh 按钮可以随机生成。在一个文本域的中央显示每个数字。对每个条目使用一个文本域。允许用户改变条目的值。单击 Find Largest Block 按钮找到包含 1 值的最大子块。高亮显示块中的数字，如图 22-14b 所示。参见 www.cs.armstrong.edu/liang/animation/FindLargestBlock.html 上提供的交互式测试。

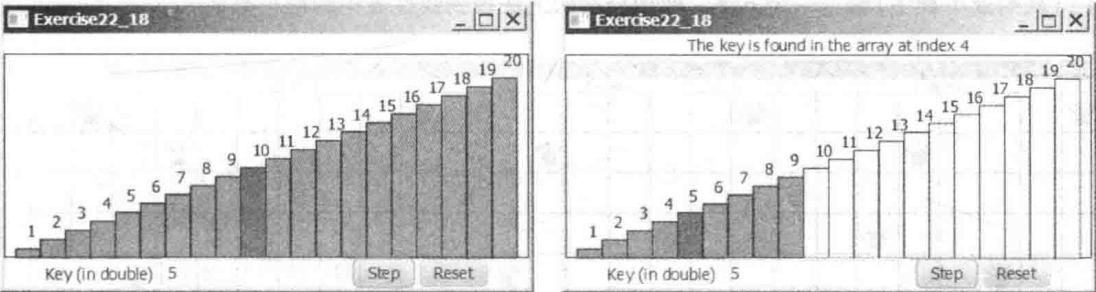


图 22-13 程序显示二分查找的动画

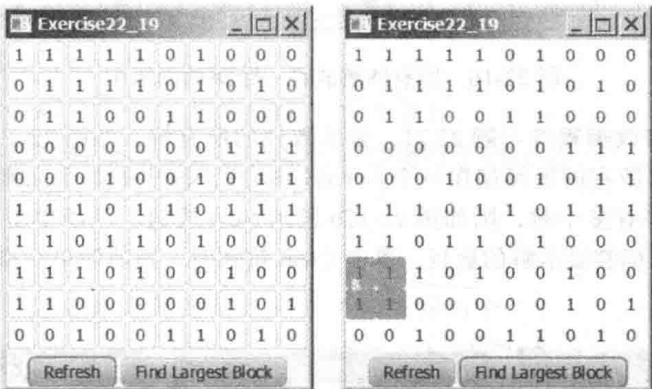


图 22-14 程序找到包含 1 的最大块

- ***22.20 (游戏：多个数独的解答) 补充材料 VI.A 给出了数独问题的完整求解。数独问题可能有多个解答。修改补充材料 VI.A 中的 Sudoku.java 显示解决方案的总数。如果多个解决方案存在，则显示两个解决方案。
- ***22.21 (游戏：数独) 补充材料 VI.C 给出了数独问题的完整求解。编写一个程序，提示用户从文本域输入数字，如图 22-15a 所示。单击 Solve 按钮显示结果，如图 22-15b ~ 图 22-15c 所示。

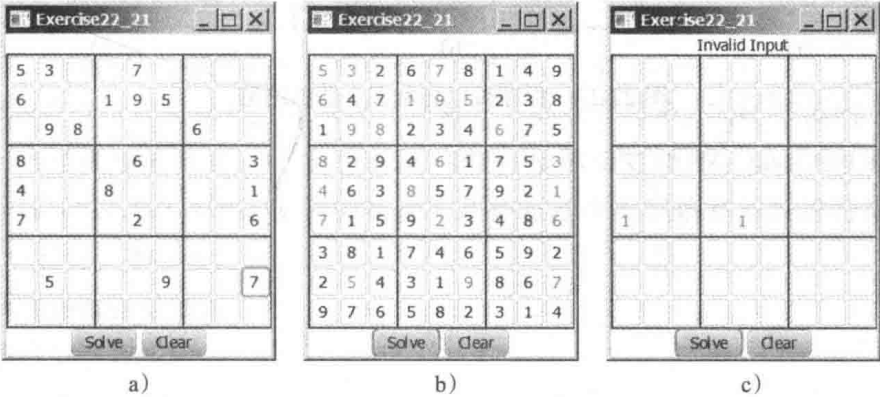


图 22-15 解决数独问题的程序

- ***22.22 (游戏: 递归数独) 为数独问题编写一个递归的解法。
- ***22.23 (游戏: 多个八皇后问题的解答) 编写一个程序, 在一个滚动面板中显示八皇后问题的所有可能解, 如图 22-16 所示。对于每个解, 使用标签标记解决方案的数字。(提示: 将所有解的面板放在一个 HBox 中, 然后将其放入一个 ScrollPane 中。)
- **22.24 (找到最小数字) 编写一个方法, 使用分而治之法找到线性表中的最小数字。

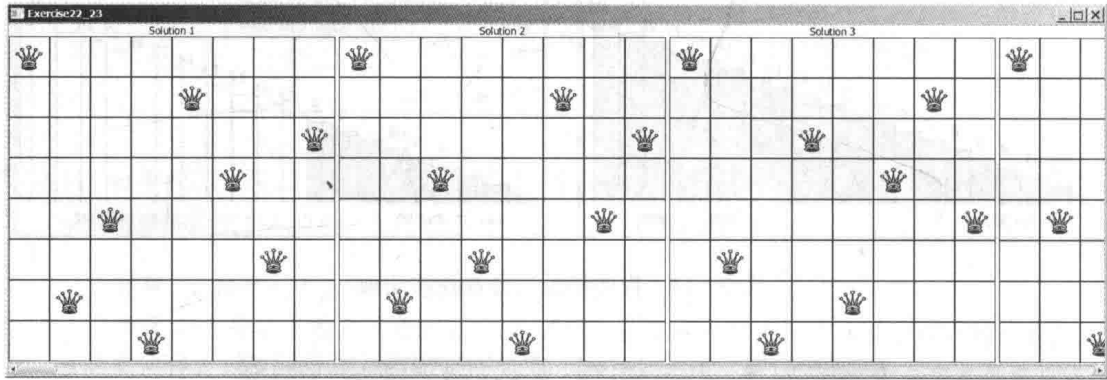


图 22-16 所有的解放在一个滚动面板中

***22.25 (游戏: 数独) 修改编程练习题 22.21, 显示数独的所有解, 如图 22-17a 所示。当单击 Solve 按钮时, 程序将所有的解保存在一个 ArrayList 中。表中的每个元素都是一个二维的 9 × 9 网格。如果程序有多个解, 则如图 22-17b 显示 Next 按钮。可以单击 Next 按钮显示下一个解, 同样有一个标签显示解的数目。单击 Clear 按钮时, 则清除单元格, 隐藏 Next 按钮, 如图 22-17c 所示。

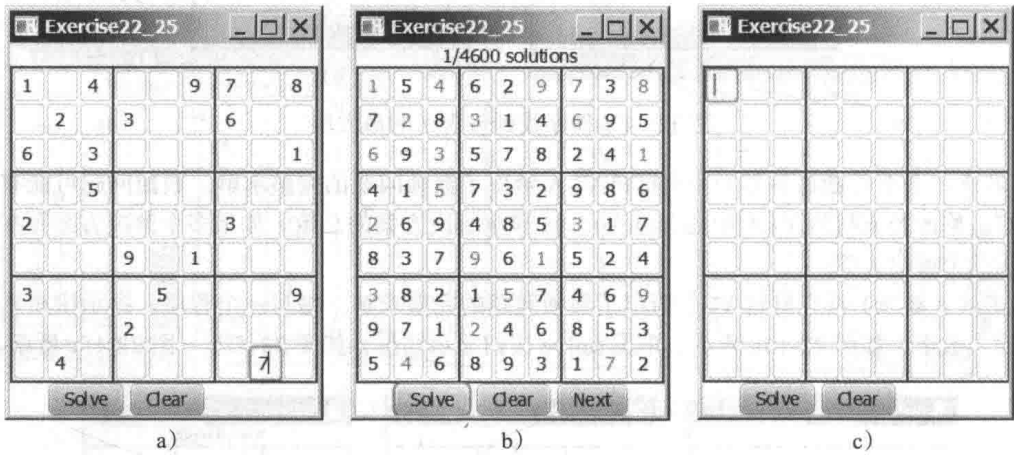


图 22-17 程序可以显示多个数独的解

排 序

23 教学目标

- 研究和分析各种排序算法的时间效率 (23.2 ~ 23.7 节)。
- 设计、实现和分析插入排序 (23.2 节)。
- 设计、实现和分析冒泡排序 (23.3 节)。
- 设计、实现和分析归并排序 (23.4 节)。
- 设计、实现和分析快速排序 (23.5 节)。
- 设计和实现一个二叉堆 (23.6 节)。
- 设计、实现和分析堆排序 (23.6 节)。
- 设计、实现和分析桶排序和基数排序 (23.7 节)。
- 设计、实现和分析对文件中大量数据的外部排序 (23.8 节)。

23.1 引言

 **要点提示：**排序算法是学习算法设计和分析的极好例子。

2007 年，当总统候选人 Barack Obama 访问 Google 公司时，Google 的 CEO Eric Schmidt 问了 Obama 一个问题，对 100 万 32 位整数排序的最有效的方式是什么 (www.youtube.com/watch?v=k4RRi_ntQc8)。Obama 回答冒泡算法将不是好的选择。他的回答正确吗？我们在本章中将考察各种排序算法，然后看看他是否正确。

在计算机科学中，排序是一个经典的主题。学习排序算法的原因有三个。

- 首先，排序算法阐明了许多解决问题的创造性的方法，并且这些方法还可用于解决其他问题。
- 其次，排序算法有助于使用选择语句、循环、方法和数组来练习基本的程序设计技术。
- 最后，排序算法是演示算法性能的极好的例子。

要排序的数据可能是整数、双精度浮点数、字符或者对象。7.11 节给出了对于数值的选择排序。选择排序算法在 19.5 节中扩展到对对象数组的排序。Java API 在 `java.util.Arrays` 和 `java.util.Collections` 类中包含了几种对基本类型值和对象进行排序的重载方法。为了简单起见，本章假定：

- 1) 要排序的数据是整数。
- 2) 数据存储在数组中。
- 3) 数据以升序排列。

排序程序可以很容易地修改为对其他类型的数据排序、以降序排列或者对 `ArrayList` 或 `LinkedList` 中的数据排序。

目前已有许多排序算法。上一章介绍了选择排序。本章将介绍插入排序、冒泡排序、归并排序、快速排序、桶排序、基数排序和外部排序。

23.2 插入排序


 **要点提示：**插入排序重复地将新的元素插入到一个排好序的子线性表中，直到整个线性表排好序。

图 23-1 描述如何用插入排序法对线性表 {2,9,5,4,8,1,6} 进行排序。

这个算法可以描述如下：

```
for (int i=1; i<list.length; i++) {
    将 list[i] 插入已排好序的子线性表中，这样 list[0..i] 也是排好序的
}
```

为了将 `list[i]` 插入 `list[0..i-1]`，需要将 `list[i]` 存储在一个名为 `currentElement` 的临时变量中。如果 `list[i-1]>currentElement`，就将 `list[i-1]` 移到 `list[i]`；如果 `list[i-2]>currentElement`，就将 `list[i-2]` 移到 `list[i-1]`，依此类推，直到 `list[i-k]<=currentElement` 或者 `k>i`（传递的是排好序的数列的第一个元素）。将 `currentElement` 赋值给 `list[i-k+1]`。例如，为了在图 23-2 的步骤 4 中将 4 插入 {2,5,9} 中，由于 `9>4`，所以把 `list[2]` (9) 移到 `list[3]`，又因为 `5>4`，所以把 `list[1]` (5) 移到 `list[2]`。最后，把 `currentElement` (4) 移到 `list[1]`。

步骤 1：最开始，排好序的子线性表只包含线性表中的第一个元素。把 9 插入到该子线性表中

2 9 5 4 8 1 6

步骤 2：排好序的子线性表为 {2,9}。把 5 插入到子线性表中

2 9 → 5 4 8 1 6

步骤 3：排好序的子线性表为 {2,5,9}。把 4 插入到子线性表中

2 5 → 9 → 4 8 1 6

步骤 4：排好序的子线性表为 {2,4,5,9}。把 8 插入到子线性表中

2 4 5 9 → 8 1 6

步骤 5：排好序的子线性表为 {2,4,5,8,9}。把 1 插入到子线性表中

2 → 4 → 5 → 8 → 9 → 1 6

步骤 6：排好序的子线性表为 {1,2,4,5,8,9}。把 6 插入到子线性表中

1 2 4 5 8 → 9 → 6

步骤 7：现在整个线性表已经排好序了

1 2 4 5 6 8 9

图 23-1 插入排序将新元素重复插入已排好序的子线性表中

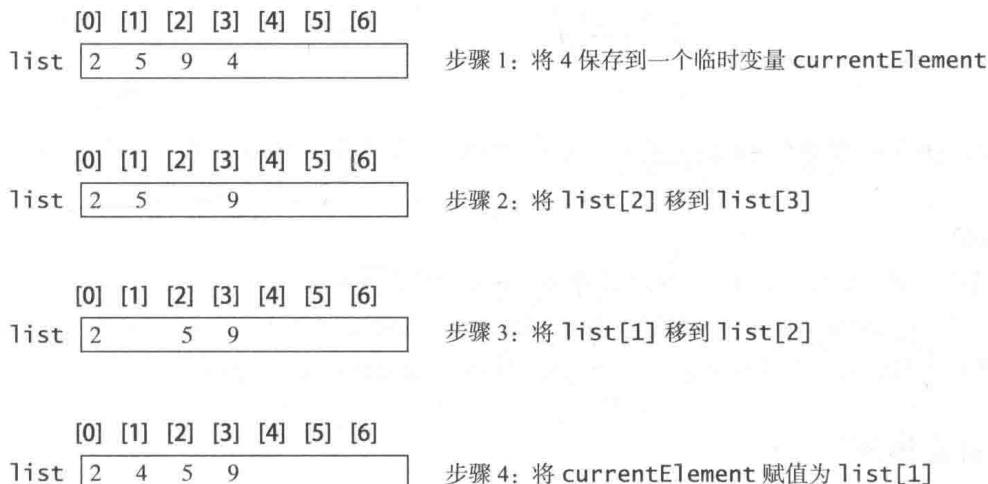


图 23-2 一个新的元素插入到排好序的子列表中

算法可以扩展和执行，如程序清单 23-1 所示。

程序清单 23-1 InsertionSort.java

```

1 public class InsertionSort {
2     /** The method for sorting the numbers */
3     public static void insertionSort(int[] list) {
4         for (int i = 1; i < list.length; i++) {
5             /** Insert list[i] into a sorted sublist list[0..i-1] so that
6              * list[0..i] is sorted. */
7             int currentElement = list[i];
8             int k;
9             for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {
10                 list[k + 1] = list[k];
11             }
12             // Insert the current element into list[k + 1]
13             list[k + 1] = currentElement;
14         }
15     }
16 }
17 }

```

`insertionSort(int[] list)` 方法对任意一个 `int` 类型元素构成的数组进行排序。该方法是用嵌套的 `for` 循环实现的。外层循环（循环控制变量 `i`）（第 4 行）的迭代是为了获取已排好序的子线性表，其范围从 `list[0]` 到 `list[i]`。内层循环（循环控制变量 `k`）将 `list[i]` 插入从 `list[0]` 到 `list[i-1]` 的子线性表中。

为了更好地理解这个方法，使用下面的语句跟踪这个方法：

```

int[] list = {1, 9, 4, 6, 5, -4};
InsertionSort.insertionSort(list);

```

这里给出的插入排序算法重复地将一个新的元素插入到一个排好序的部分数组中，直到整个数组排好序。在第 k 次迭代中，为了将一个元素插入到一个大小为 k 的数组中，将进行 k 次比较来找到插入的位置，还要进行 k 次的移动来插入元素。使用 $T(n)$ 表示插入排序的复杂度， c 表示诸如每次迭代中的赋值和额外的比较的操作总数，则

$$\begin{aligned}
 T(n) &= (2+c) + (2 \times 2+c) + \cdots + (2 \times (n-1)+c) \\
 &= 2(1+2+\cdots+n-1) + c(n-1)
 \end{aligned}$$

$$\begin{aligned}
 &= 2 \frac{(n-1)n}{2} + cn - c = n^2 - n + cn - c \\
 &= O(n^2)
 \end{aligned}$$

因此, 插入排序算法的复杂度为 $O(n^2)$ 。因此, 选择排序和插入排序具有同样的时间复杂度。

复习题

- 23.1 描述插入排序是如何工作的。插入排序的时间复杂度为多少?
 23.2 使用图 23-1 作为一个例子来演示如何在 $\{45, 11, 50, 59, 60, 2, 4, 7, 10\}$ 上应用插入排序。
 23.3 如果一个线性表已经排好了, `insertionSort` 方法将执行多少次比较?

23.3 冒泡排序

要点提示: 冒泡排序算法多次遍历数组, 在每次遍历中连续比较相邻的元素, 如果元素没有按照顺序排列, 则互换它们的值。

冒泡排序算法需要遍历几次数组。在每次遍历中, 比较连续相邻的元素。如果某一对元素是降序, 则互换它们的值; 否则, 保持不变。由于较小的值像“气泡”一样逐渐浮向顶部, 而较大的值沉向底部, 所以称这种技术为冒泡排序 (bubble sort) 或下沉排序 (sinking sort)。第一次遍历之后, 最后一个元素成为数组中的最大数。第二次遍历之后, 倒数第二个元素成为数组中的第二大数。整个过程持续到所有元素都已排好序。

图 23-3a 给出由 6 个元素 (2 9 5 4 8 1) 构成的数组经过第一次冒泡排序的遍历情况。首先比较第一对元素 (2 和 9), 因为这两个数已经是顺序排列的, 所以不需要交换。接着比较第二对元素 (9 和 5), 因为 9 大于 5, 所以交换 9 和 5。然后比较第三对元素 (9 和 4), 并交换 9 和 4。再比较第四对元素 (9 和 8), 并交换 9 和 8。最后比较第五对元素 (9 和 1), 并交换 9 和 1。在图 23-3 中, 被比较的数对被突出显示, 已经排好序的数字用斜体表示。

2 9 5 4 8 1	2 5 4 8 1 9	2 4 5 1 8 9	2 4 1 5 8 9	1 2 4 5 8 9
2 5 9 4 8 1	2 4 5 8 1 9	2 4 5 1 8 9	2 1 4 5 8 9	
2 5 4 9 8 1	2 4 5 8 1 9	2 4 1 5 8 9		
2 5 4 8 9 1	2 4 5 1 8 9			
2 5 4 8 1 9				
a) 第 1 次遍历	b) 第 2 次遍历	c) 第 3 次遍历	d) 第 4 次遍历	e) 第 5 次遍历

图 23-3 每次遍历都依次对元素对进行比较和排序

经过第 1 次遍历后, 最大数 (9) 放置在数组的末尾。在如图 23-3b 所示的第 2 次遍历中, 依次对元素进行比较和排序。因为数组中的最后一个元素已经是最大的, 所以不必考虑最后一对元素。在如图 23-3c 所示的第 3 次遍历中, 因为最后两个元素已排好序, 所以对除了它们之外的元素对进行顺序比较和排序。因此, 在第 k 次遍历时, 不需要考虑最后 $k-1$ 个元素, 因为它们已经排好序了。

冒泡排序算法在程序清单 23-2 中描述。

程序清单 23-2 Bubble Sort Algorithm

```

1 for (int k = 1; k < list.length; k++) {
2     // Perform the kth pass

```

```

3   for (int i = 0; i < list.length - k; i++) {
4       if (list[i] > list[i + 1])
5           swap list[i] with list[i + 1];
6   }
7   }

```

注意到如果在某次遍历中没有发生交换，那么就不必进行下一次遍历，因为所有的元素都已经排好序了。使用该特性可以改进上面程序清单 23-2 中的算法，如程序清单 23-3 所示。

程序清单 23-3 Improved Bubble Sort Algorithm

```

1   boolean needNextPass = true;
2   for (int k = 1; k < list.length && needNextPass; k++) {
3       // Array may be sorted and next pass not needed
4       needNextPass = false;
5       // Perform the kth pass
6       for (int i = 0; i < list.length - k; i++) {
7           if (list[i] > list[i + 1]) {
8               swap list[i] with list[i + 1];
9               needNextPass = true; // Next pass still needed
10          }
11      }
12  }

```

算法可以在程序清单 23-4 中实现。

程序清单 23-4 BubbleSort.java

```

1   public class BubbleSort {
2       /** Bubble sort method */
3       public static void bubbleSort(int[] list) {
4           boolean needNextPass = true;
5
6           for (int k = 1; k < list.length && needNextPass; k++) {
7               // Array may be sorted and next pass not needed
8               needNextPass = false;
9               for (int i = 0; i < list.length - k; i++) {
10                  if (list[i] > list[i + 1]) {
11                      // Swap list[i] with list[i + 1]
12                      int temp = list[i];
13                      list[i] = list[i + 1];
14                      list[i + 1] = temp;
15
16                      needNextPass = true; // Next pass still needed
17                  }
18              }
19          }
20      }
21
22      /** A test method */
23      public static void main(String[] args) {
24          int[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
25          bubbleSort(list);
26          for (int i = 0; i < list.length; i++)
27              System.out.print(list[i] + " ");
28      }
29  }

```

-2 1 2 2 3 3 5 6 12 14

在最佳情况下，冒泡排序算法只需要一次遍历就能确定数组已排好序，不需要进行下一次遍历。由于第一次遍历的比较次数为 $n-1$ ，因此在最佳情况下，冒泡排序的时间为 $O(n)$ 。

在最差情况下,冒泡排序算法需要进行 $n-1$ 次遍历。第 1 次遍历需要 $n-1$ 次比较;第 2 次遍历需要 $n-2$ 次比较;依此进行,最后一次遍历需要 1 次比较。因此,比较的总数为:

$$\begin{aligned} & (n-1) + (n-2) + \cdots + 2 + 1 \\ &= \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2) \end{aligned}$$

因此,在最差情况下,冒泡排序的时间为 $O(n^2)$ 。

复习题

- 23.4 描述冒泡排序法是如何工作的。冒泡排序的时间复杂度是多少?
 23.5 使用图 23-3 作为一个例子,演示如何将冒泡排序应用在 $\{45, 11, 50, 59, 60, 2, 4, 7, 10\}$ 上。
 23.6 如果一个线性表已经排好了, bubbleSort 方法还需要进行多少次比较?

23.4 归并排序

要点提示: 归并排序算法将数组分为两半,对每部分递归地应用归并排序。在两部分都排好后,对它们进行归并。

归并排序的算法在程序清单 23-5 中给出。

程序清单 23-5 归并排序算法

```
1 public static void mergeSort(int[] list) {
2     if (list.length > 1) {
3         mergeSort(list[0 ... list.length / 2]);
4         mergeSort(list[list.length / 2 + 1 ... list.length]);
5         merge list[0 ... list.length / 2] with
6             list[list.length / 2 + 1 ... list.length];
7     }
8 }
```

图 23-4 演示了对由 8 个元素 (2 9 5 4 8 1 6 7) 构成的数组进行的归并排序。原始数组分为 (2 9 5 4) 和 (8 1 6 7) 两组。对这两个子数组递归地应用归并排序,将 (2 9 5 4) 分为 (2 9) 和 (5 4),并将 (8 1 6 7) 分为 (8 1) 和 (6 7)。继续进行这个过程直到子数组只包含一个元素为止。例如,将数组 (2 9) 分为 (2) 和 (9)。由于 (2) 包含的是单一元素,所以它不能再细分了。现在,将 (2) 和 (9) 归并为一个新的有序数组 (2 9),将 (5) 和 (4) 归并为一个新的有序数组 (4 5)。然后将 (2 9) 和 (4 5) 归并为一个新的有序数组 (2 4 5 9),最后将 (2 4 5 9) 和 (1 6 7 8) 归并为一个新的有序数组 (1 2 4 5 6 7 8 9)。

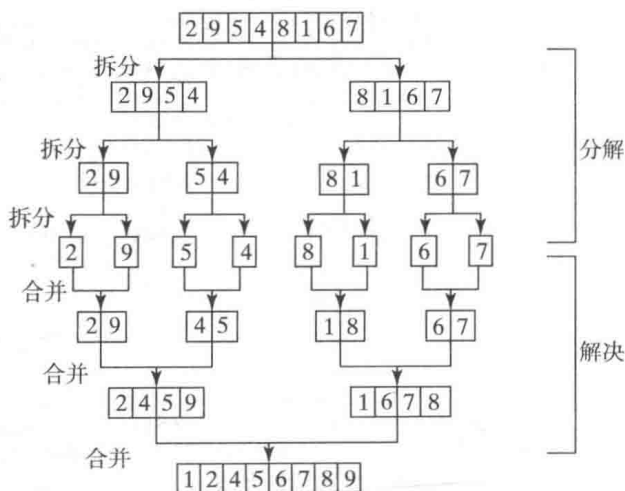


图 23-4 归并排序使用分而治之法对数组排序

递归调用持续将数组划分为子数组,直到每个子数组只包含一个元素。然后,该算法将这些小的子数组归并为稍大的有序子数组,直到最后形成一个有序的数组。

归并排序算法在程序清单 23-6 中实现。

程序清单 23-6 MergeSort.java

```
1 public class MergeSort {
2     /** The method for sorting the numbers */
3     public static void mergeSort(int[] list) {
4         if (list.length > 1) {
5             // Merge sort the first half
6             int[] firstHalf = new int[list.length / 2];
7             System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
8             mergeSort(firstHalf);
9
10            // Merge sort the second half
11            int secondHalfLength = list.length - list.length / 2;
12            int[] secondHalf = new int[secondHalfLength];
13            System.arraycopy(list, list.length / 2,
14                secondHalf, 0, secondHalfLength);
15            mergeSort(secondHalf);
16
17            // Merge firstHalf with secondHalf into list
18            merge(firstHalf, secondHalf, list);
19        }
20    }
21
22    /** Merge two sorted lists */
23    public static void merge(int[] list1, int[] list2, int[] temp) {
24        int current1 = 0; // Current index in list1
25        int current2 = 0; // Current index in list2
26        int current3 = 0; // Current index in temp
27
28        while (current1 < list1.length && current2 < list2.length) {
29            if (list1[current1] < list2[current2])
30                temp[current3++] = list1[current1++];
31            else
32                temp[current3++] = list2[current2++];
33        }
34
35        while (current1 < list1.length)
36            temp[current3++] = list1[current1++];
37
38        while (current2 < list2.length)
39            temp[current3++] = list2[current2++];
40    }
41
42    /** A test method */
43    public static void main(String[] args) {
44        int[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
45        mergeSort(list);
46        for (int i = 0; i < list.length; i++)
47            System.out.print(list[i] + " ");
48    }
49 }
```

方法 mergeSort (第 3 ~ 20 行) 创建一个新数组 firstHalf, 该数组是 list 前半部分的一个副本 (第 7 行)。算法在 firstHalf 上递归地调用 mergeSort (第 8 行)。firstHalf 的长度为 list.length/2, 而 secondHalf 的长度为 list.length-list.length/2。创建的新数组 secondHalf 包含原始数组 list 的后半部分。算法在 secondHalf 上递归地调用 mergeSort (第 15 行)。在 firstHalf 和 secondHalf 都排好序之后, 将它们归并成一个新的有序数组 list (第 18 行)。这样, 数组 list 就排好序了。

方法 merge (第 23 ~ 40 行) 归并两个有序数组 list1 和 list2 为一个临时数组 temp。

current1 和 current2 指向 list1 和 list2 中要考虑的当前元素 (第 24 ~ 26 行)。该方法重复比较 list1 和 list2 中的当前元素, 并将较小的一个元素移动到 temp 中。如果较小元素在 list1 中, current1 增加 1 (第 30 行); 如果较小元素在 list2 中, current2 增加 1 (第 32 行)。最后, 其中一个数组中的所有元素都被移动到 temp 中。如果 list1 中仍有未移动的元素, 就将它们复制到 temp 中 (第 35 ~ 36 行)。如果 list2 中仍有未移动的元素, 就将它们复制到 temp 中 (第 38 ~ 39 行)。

图 23-5 演示了如何将两个数组 list1 (2 4 5 9) 和 list2 (1 6 7 8) 进行归并。初始状态时, 要考虑的数组中的两个当前元素是 2 和 1。比较这两个数, 并将较小元素 1 移到 temp 中, 如图 23-5a 所示。current2 和 current3 增加 1。继续比较这两个数组中的当前元素, 并将较小数移动到 temp 中, 直到其中一个数组移动完毕。如图 23-5b 所示, list2 中的所有元素都被移动到 temp 中, 而且 current1 指向 list1 中的元素 9。将 9 复制到 temp 中, 如图 23-5c 所示。

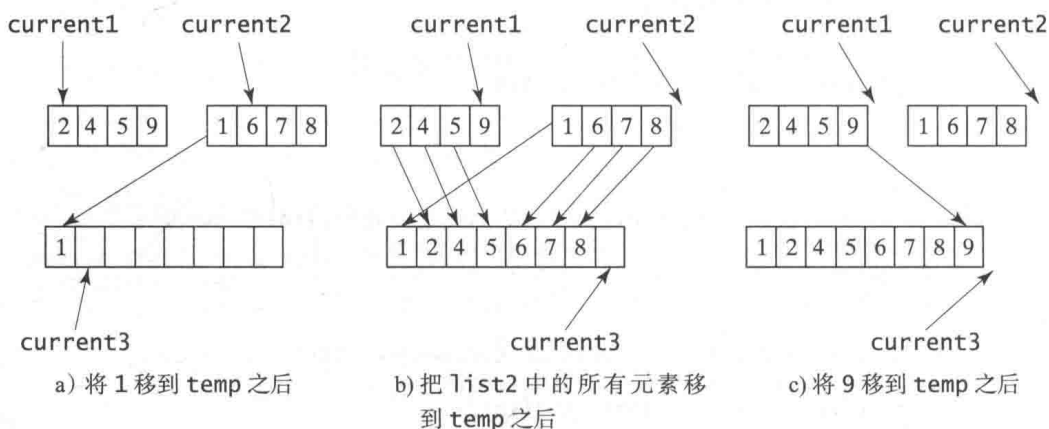


图 23-5 将两个有序数组归并为一个有序数组

MergeSort 方法在分解过程中创建两个临时数组 (第 6 和 12 行), 将数组的前半部分和后半部分复制到临时数组中 (第 7 和 13 行), 对临时数组排序 (第 8 和 15 行), 然后将它们归并到原始数组中 (第 18 行), 如图 23-6 所示。可以改写该代码, 递归地对数组的前半部分和后半部分进行排序, 而不创建新的临时数组, 然后把两个数组归并到一个临时数组中并将它的内容复制到初始数组中, 如图 23-6b 所示。这个留作编程练习题 23.20。

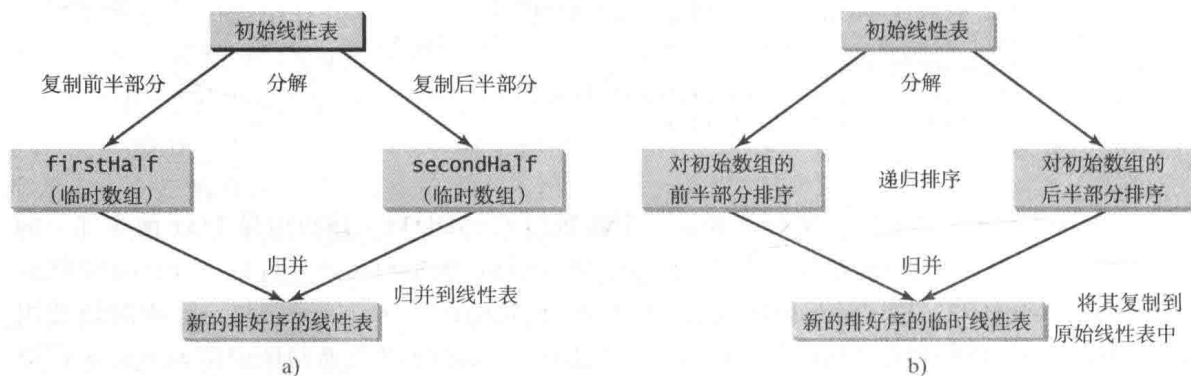


图 23-6 创建临时数组以支持归并排序

注意: 归并排序可以使用并行处理高效执行。参见 30.16 节中归并排序的并行实现。

假设 $T(n)$ 表示使用归并排序对由 n 个元素构成的数组进行排序所需的时间。不失一般性, 假设 n 是 2 的幂。归并排序算法将数组分为两个子数组, 使用同样的算法对子数组进行递归排序, 然后将子数组进行归并。因此

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \text{mergetime}$$

第一项是对数组的前半部分排序所需的时间, 而第二项是对数组的后半部分排序所需的时间。要归并两个子数组, 最多需要 $n-1$ 次比较来比较两个子数组中的元素, 以及 n 次移动将元素移到临时数组中。因此, 总时间为 $2n-1$ 。因此

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 2n - 1 = O(n \log n)$$

归并排序的复杂度为 $O(n \log n)$ 。该算法优于选择排序、插入排序和冒泡排序, 因为这些排序算法的复杂度为 $O(n^2)$ 。java.util.Arrays 类中的 sort 方法是使用归并排序算法的变体来实现的。

✓ 复习题

- 23.7 描述归并排序是如何工作的。归并排序的时间复杂度为多少?
- 23.8 以图 23-4 为例, 演示如何在 {45, 11, 50, 59, 60, 2, 4, 7, 10} 上使用归并排序。
- 23.9 如果程序清单 23-6 中的第 6 ~ 15 行被下面代码替代, 会有什么错误?

```
// Merge sort the first half
int[] firstHalf = new int[list.length / 2 + 1];
System.arraycopy(list, 0, firstHalf, 0, list.length / 2 + 1);
mergeSort(firstHalf);

// Merge sort the second half
int secondHalfLength = list.length - list.length / 2 - 1;
int[] secondHalf = new int[secondHalfLength];
System.arraycopy(list, list.length / 2 + 1,
    secondHalf, 0, secondHalfLength);
mergeSort(secondHalf);
```

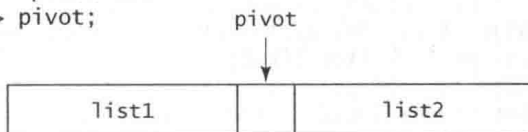
23.5 快速排序

🔑 **要点提示:** 快速排序工作机制如下, 该算法在数组中选择一个称为主元 (pivot) 的元素, 将数组分为两部分, 使得第一部分中的所有元素都小于或等于主元, 而第二部分中的所有元素都大于主元。对第一部分递归地应用快速排序算法, 然后对第二部分递归地应用快速排序算法。

快速排序是由 C. A. R. Hoare 于 1962 年开发的, 该算法在程序清单 23-7 中描述。

程序清单 23-7 Quick Sort Algorithm

```
1 public static void quickSort(int[] list) {
2     if (list.length > 1) {
3         select a pivot;
4         partition list into list1 and list2 such that
5             all elements in list1 <= pivot and
6             all elements in list2 > pivot;
7         quickSort(list1);
8         quickSort(list2);
9     }
10 }
```



该算法的每次划分都将主元放在了恰当的位置。主元的选择会影响算法的性能。在理想情况下，应该选择能平均划分两部分的主元。为了简单起见，假定将数组的第一个元素选为主元。（编程练习题 23.4 提出了选择主元的一个替代的策略。）

图 23-7 演示了如何使用快速排序算法对数组 (5 2 9 3 8 4 0 1 6 7) 排序。选择第一个元素 5 作为主元将该数组划分为两部分，如图 23-7b 所示。突出显示的主元放在数组的恰当位置。分别对两个子数组 (4 2 1 3 0) 和 (8 9 6 7) 应用快速排序。主元 4 将 (4 2 1 3 0) 仅划分为一个数组 (0 2 1 3)，如图 23-7c 所示。然后对 (0 2 1 3) 应用快速排序。主元 0 将 (0 2 1 3) 也仅分为一个数组 (2 1 3)，如图 23-7d 所示。再对 (2 1 3) 应用快速排序。主元 2 将 (2 1 3) 分为 (1) 和 (3)，如图 23-7e 所示。再对 (1) 应用快速排序。由于该数组只包含一个元素，所以无须进一步划分。

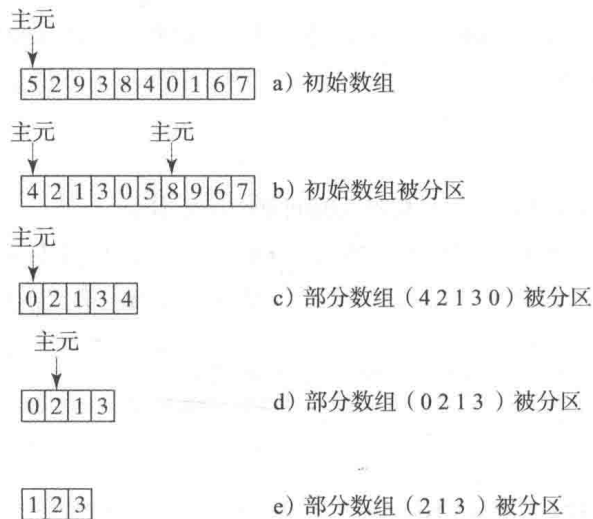


图 23-7 快速排序算法递归地应用在部分数组上

快速排序算法在程序清单 23-8 中实现。类中有两个重载的 `quickSort` 方法。第一个方法（第 2 行）用来对数组进行排序。第二个是一个辅助方法（第 6 行），用于对特定范围内的子数组进行排序。

程序清单 23-8 QuickSort.java

```

1 public class QuickSort {
2     public static void quickSort(int[] list) {
3         quickSort(list, 0, list.length - 1);
4     }
5
6     public static void quickSort(int[] list, int first, int last) {
7         if (last > first) {
8             int pivotIndex = partition(list, first, last);
9             quickSort(list, first, pivotIndex - 1);
10            quickSort(list, pivotIndex + 1, last);
11        }
12    }
13
14    /** Partition the array list[first..last] */
15    public static int partition(int[] list, int first, int last) {
16        int pivot = list[first]; // Choose the first element as the pivot
17        int low = first + 1; // Index for forward search
18        int high = last; // Index for backward search

```

```
19
20 while (high > low) {
21     // Search forward from left
22     while (low <= high && list[low] <= pivot)
23         low++;
24
25     // Search backward from right
26     while (low <= high && list[high] > pivot)
27         high--;
28
29     // Swap two elements in the list
30     if (high > low) {
31         int temp = list[high];
32         list[high] = list[low];
33         list[low] = temp;
34     }
35 }
36
37 while (high > first && list[high] >= pivot)
38     high--;
39
40 // Swap pivot with list[high]
41 if (pivot > list[high]) {
42     list[first] = list[high];
43     list[high] = pivot;
44     return high;
45 }
46 else {
47     return first;
48 }
49 }
50
51 /** A test method */
52 public static void main(String[] args) {
53     int[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
54     quickSort(list);
55     for (int i = 0; i < list.length; i++)
56         System.out.print(list[i] + " ");
57 }
58 }
```

-2 1 2 2 3 3 5 6 12 14

方法 `partition` (第 15 ~ 49 行) 使用主元划分数组 `list[first..last]`。将子数组的第一个元素选为主元 (第 16 行)。在初始情况下, `low` 指向子数组中的第二个元素 (第 17 行), 而 `high` 指向子数组中的最后一个元素 (第 18 行)。

方法在数组中从左侧开始查找第一个大于主元的元素 (第 22 ~ 23 行), 然后从数组右侧开始查找第一个小于或等于主元的元素 (第 26 ~ 27 行), 最后交换这两个元素。在 `while` 循环中重复相同的查找和交换操作, 直到所有元素都查找完为止 (第 20 ~ 35 行)。

如果主元被移动, 方法返回将子数组分为两部分的主元的新下标 (第 44 行); 否则, 返回主元的原始下标 (第 47 行)。

图 23-8 演示了如何划分数组 (5 2 9 3 8 4 0 1 6 7)。选择第一个元素 5 作为主元。在初始状态时, `low` 是指向元素 2 的下标, 而 `high` 是指向元素 7 的下标, 如图 23-8a 所示。推进下标 `low` 查找第一个大于主元的元素 (9), 然后从下标 `high` 往回推查找第一个小于或等于主元的元素 (1), 如图 23-8b 所示。交换 9 和 1, 如图 23-8c 所示。继续查找, 移动 `low` 使其指向元素 8, 移动 `high` 使其指向元素 0, 如图 23-8d 所示。交换 8 和 0, 如图 23-8e 所示。

继续移动 low 直到它超过 high，如图 23-8f 所示。现在所有的元素都检查过了，交换主元与下标 high 处的元素 4。最终的划分情况如图 23-8g 所示。当方法结束的时候，返回主元的下标。

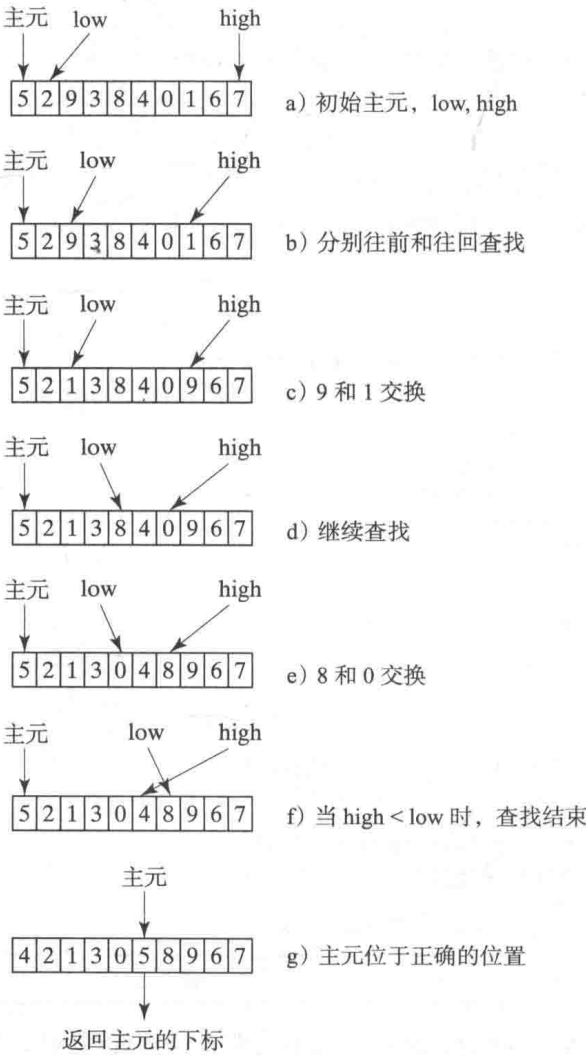


图 23-8 partition 方法在主元放置在正确的位置后返回主元的下标

在最差情况下，划分由 n 个元素构成的数组需要进行 n 次比较和 n 次移动。因此，划分所需时间为 $O(n)$ 。

在最差情况下，每次主元会将数组划分为一个大的子数组和一个空数组。这个大的子数组的规模是在上次划分的子数组的规模上减 1。该算法需要 $(n-1) + (n-2) + \cdots + 2 + 1 = O(n^2)$ 时间。

在最佳情况下，每次主元将数组划分为规模大致相等的两部分。设 $T(n)$ 表示使用快速排序算法对包含 n 个元素的数组排序所需的时间，因此

在两个子数组上面进行递归的快速排序

分区的时间

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n$$

和归并排序的分析相似，快速排序的 $T(n) = O(n \log n)$ 。

在平均情况下，每次主元不会将数组分为规模相等的两部分或是一个空的部分。从统计上说，两部分的规模会非常接近。因此，平均时间为 $O(n \log n)$ 。精确的平均情况分析已经超出了本书的范围。

归并排序和快速排序都使用了分而治之法。对于归并排序，大量的工作是将两个子线性表进行归并，归并是在子线性表都排好序后进行的。对于快速排序，大量的工作是将线性表划分为两个子线性表，划分是在子线性表排好序前进行的。在最差情况下，归并排序的效率高于快速排序，但是，在平均情况下，两者的效率相同。归并排序在归并两个子数组时需要一个临时数组，而快速排序不需要额外的数组空间。因此，快速排序的空间效率高于归并排序。

✓ 复习题

23.10 描述快速排序是如何工作的。快速排序的时间复杂度是多少？

23.11 为什么快速排序比归并排序的空间效率更高？

23.12 以图 23-7 为例，演示如何在 {45, 11, 50, 59, 60, 2, 4, 7, 10} 上面应用快速排序。

23.6 堆排序

要点提示：堆排序使用的是二叉堆。它首先将所有的元素添加到一个堆上，然后不断移除最大的元素以获得一个排好序的线性表。

堆排序 (heap sort) 使用二叉堆，它是一棵完全二叉树。二叉树是一种层次体系结构。它可能是空的，也可能包含一个称为根 (root) 的元素以及称为左子树 (left subtree) 和右子树 (right subtree) 的两棵不同的二叉树。一条路径的长度 (length) 是指这条路径上的边数。一个结点的深度 (depth) 是指从根结点到该结点的路径的长度。

二叉堆 (binary heap) 是一棵具有以下属性的二叉树：

- 形状属性：它是一棵完全二叉树。
- 堆属性：每个结点大于或等于它的任意一个孩子。

如果一棵二叉树的每一层都是满的，或者最后一层可以不填满并且最后一层的叶子都是靠左放置的，那么这棵二叉树就是完全的 (complete)。例如，在图 23-9 中，a) 和 b) 中的二叉树都是完全的，但是 c) 和 d) 中的二叉树都不是完全的。而且，a 中的二叉树是一个堆，但是 b 中的二叉树不是堆，因为根 (39) 小于它的右孩子 (42)。

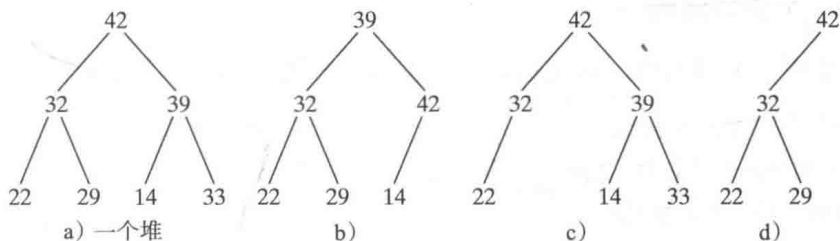


图 23-9 一个二叉堆是一种特殊的完全二叉树

注意：堆是一个在计算机科学中具有许多含义的词汇。本章中，堆表示一个二叉堆。

教学注意：堆在插入键值和删除根结点时，执行效率很高。在链接 www.cs.armstrong.edu/liang/animation/web/Heap.html 上可以通过一个交互式的演示看到堆是如何工作的，如图 23-10 所示。

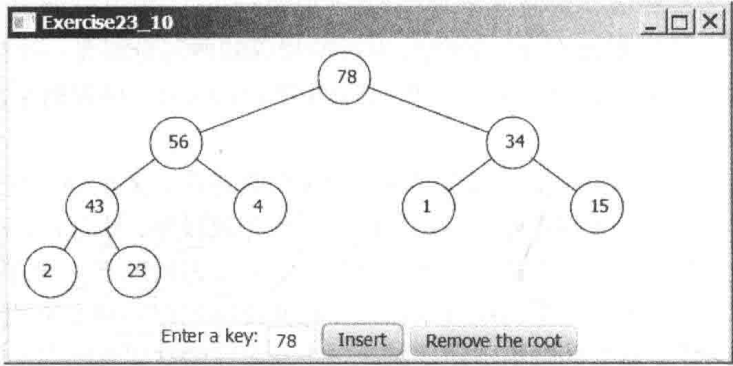


图 23-10 堆的动画工具允许可视化地插入键值以及删除根结点

23.6.1 堆的存储

如果堆的大小是事先知道的，那么可以将堆存储在一个 ArrayList 或一个数组中。图 23-11a 中的堆可以使用图 23-11b 中的数组来存储。树根在位置 0 处，它的两个子结点在位置 1 和位置 2 处。对于位置 i 处的结点，它的左子结点在位置 $2i+1$ 处，它的右子结点在位置 $2i+2$ 处，而它的父结点在位置 $(i-1)/2$ 处。例如，元素 39 的结点在位置 4 处，因此，它的左子结点（元素 14）在位置 9 处（ $2 \times 4+1$ ），它的右子结点（元素 33）在位置 10 处（ $2 \times 4+2$ ），而它的父结点（元素 42）在位置 1 处（ $(4-1)/2$ ）。

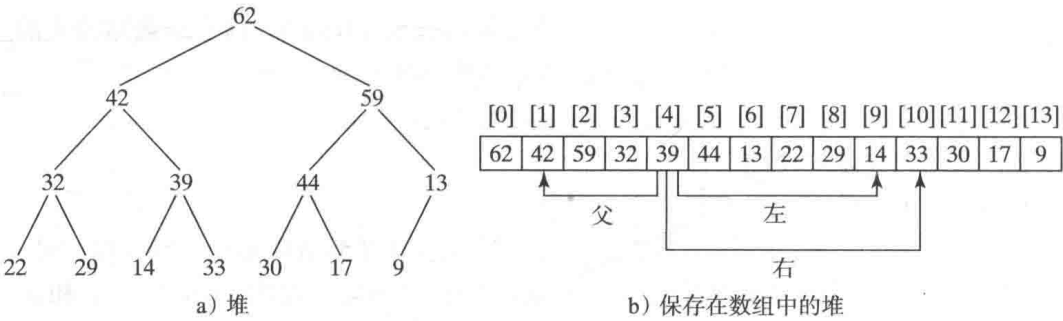


图 23-11 可以使用数组实现二叉堆

23.6.2 添加一个新的结点

为了给堆添加一个新结点，首先将它添加到堆的末尾，然后按如下方式重建这棵树：

将最后一个结点作为当前结点；

```
while ( 当前结点大于它的父结点 ){
```

将当前结点和它的父结点交换；

现在当前结点往上面进了一个层次；

```
}
```

假设这个堆被初始化为空的。在以 3、5、1、19、11 和 22 的顺序添加数字之后，这个堆如图 23-12 所示。

现在考虑向堆中添加数字 88。将新结点 88 放在树的末尾，如图 23-13a 所示。互换 88

和 19，如图 23-13b 所示。互换 88 和 22，如图 23-13c 所示。

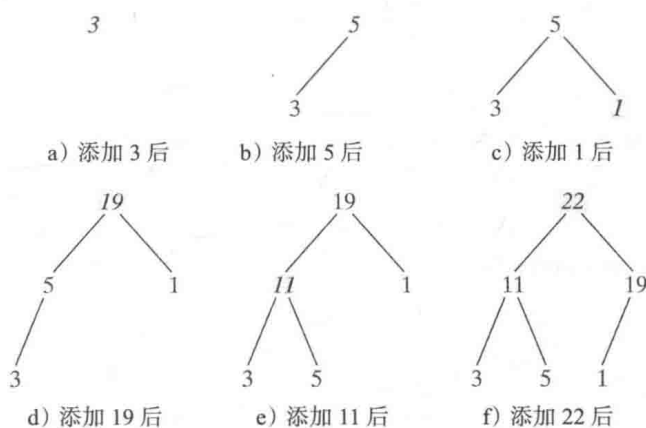


图 23-12 将元素 3、5、1、19、11 和 22 插入堆中

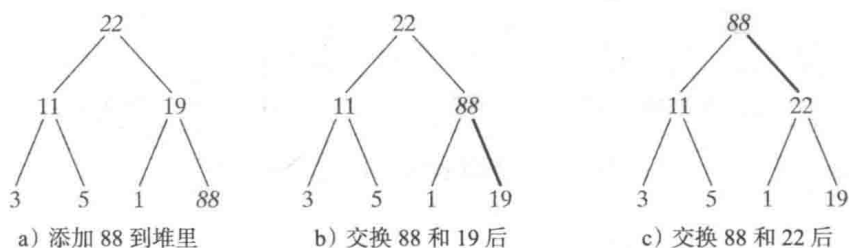


图 23-13 在添加一个元素之后重建这个堆

23.6.3 删除根结点

经常需要从堆中删除最大的元素，也就是这个堆中的根结点。在删除根结点之后，就必须重建这棵树以保持堆的属性。重建该树的算法如下所示：

```

用最后一个结点替换根结点；
让根结点成为当前结点；
while (当前结点具有子结点并且当前结点小于它的子结点){
    将当前结点和它的较大子结点交换；
    现在当前结点往下面退了一个层次；
}

```

图 23-14 给出了从图 23-11a 中删除根结点 62 之后重建堆的过程。将最后的结点 9 移到根结点处，如图 23-14a 所示。互换 9 和 59，如图 23-14b 所示。互换 9 和 44，如图 23-14c 所示。互换 9 和 30，如图 23-14d 所示。

图 23-15 给出了从图 23-14d 中删除根结点 59 之后重建堆的过程。将最后的结点 17 移到根结点处，如图 23-15a 所示。互换 17 和 44，如图 23-15b 所示。互换 17 和 30，如图 23-15c 所示。

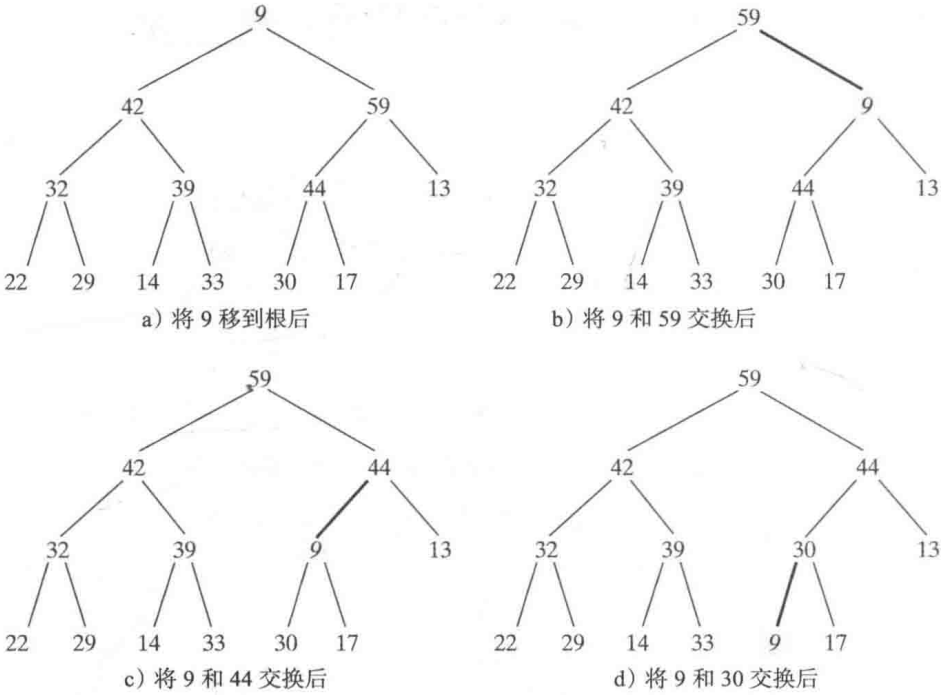


图 23-14 在删除根结点 62 之后重建堆

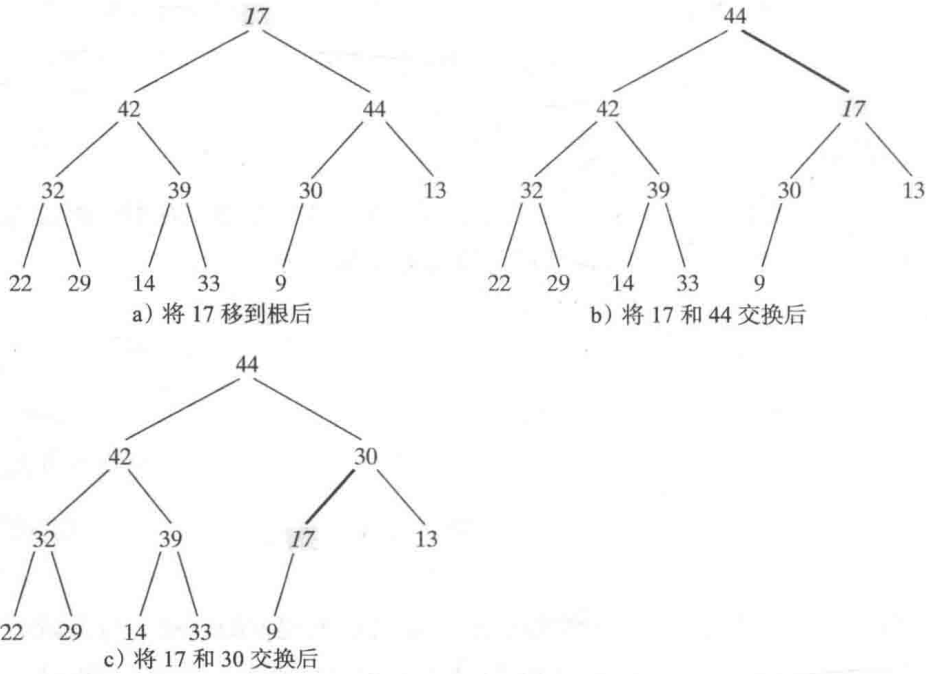


图 23-15 在删除根结点 59 之后重建堆

23.6.4 Heap 类

现在，可以设计和实现 Heap 类了。其类图如图 23-16 所示。它的实现在程序清单 23-9 中给出。

Heap<E extends Comparable<E>>	
-list: java.util.ArrayList<E>	
+Heap() +Heap(objects: E[]) +add(newObject: E): void +remove(): E +getSize(): int	创建一个默认的空的堆 创建一个具有指定对象的堆 添加一个新的对象到堆中 将根结点从堆中删除并且返回该结点 返回堆的大小

图 23-16 Heap 类提供了处理堆的操作

程序清单 23-9 Heap.java

```

1  public class Heap<E extends Comparable<E>> {
2      private java.util.ArrayList<E> list = new java.util.ArrayList<>();
3
4      /** Create a default heap */
5      public Heap() {
6      }
7
8      /** Create a heap from an array of objects */
9      public Heap(E[] objects) {
10         for (int i = 0; i < objects.length; i++)
11             add(objects[i]);
12     }
13
14     /** Add a new object into the heap */
15     public void add(E newObject) {
16         list.add(newObject); // Append to the heap
17         int currentIndex = list.size() - 1; // The index of the last node
18
19         while (currentIndex > 0) {
20             int parentIndex = (currentIndex - 1) / 2;
21             // Swap if the current object is greater than its parent
22             if (list.get(currentIndex).compareTo(
23                 list.get(parentIndex)) > 0) {
24                 E temp = list.get(currentIndex);
25                 list.set(currentIndex, list.get(parentIndex));
26                 list.set(parentIndex, temp);
27             }
28             else
29                 break; // The tree is a heap now
30
31             currentIndex = parentIndex;
32         }
33     }
34
35     /** Remove the root from the heap */
36     public E remove() {
37         if (list.size() == 0) return null;
38
39         E removedObject = list.get(0);
40         list.set(0, list.get(list.size() - 1));
41         list.remove(list.size() - 1);
42
43         int currentIndex = 0;
44         while (currentIndex < list.size()) {
45             int leftChildIndex = 2 * currentIndex + 1;
46             int rightChildIndex = 2 * currentIndex + 2;
47

```

```

48     // Find the maximum between two children
49     if (leftChildIndex >= list.size()) break; // The tree is a heap
50     int maxIndex = leftChildIndex;
51     if (rightChildIndex < list.size()) {
52         if (list.get(maxIndex).compareTo(
53             list.get(rightChildIndex)) < 0) {
54             maxIndex = rightChildIndex;
55         }
56     }
57
58     // Swap if the current node is less than the maximum
59     if (list.get(currentIndex).compareTo(
60         list.get(maxIndex)) < 0) {
61         E temp = list.get(maxIndex);
62         list.set(maxIndex, list.get(currentIndex));
63         list.set(currentIndex, temp);
64         currentIndex = maxIndex;
65     }
66     else
67         break; // The tree is a heap
68 }
69
70 return removedObject;
71 }
72
73 /** Get the number of nodes in the tree */
74 public int getSize() {
75     return list.size();
76 }
77 }

```

堆在内部是使用数组线性表来表示的（第 2 行）。可以将它改为其他的数据结构，但是 Heap 类的合约保持不变。

方法 add(E newObject)（第 15 ~ 33 行）将一个对象追加到树中，如果该对象大于它的父结点，就互换它们。此过程持续到该新对象成为根结点，或者新对象不大于它的父结点。

方法 remove()（第 36 ~ 71 行）删除并返回根结点。为保持堆的特征，该方法将最后的对象移到根结点处，如果该对象小于它的较大的子结点，就互换它们。此过程持续到最后一个对象成为叶子结点，或者该对象不小于它的子结点。

23.6.5 使用 Heap 类进行排序

要使用堆对数组排序，应首先使用 Heap 类创建一个对象，使用 add 方法将所有元素添加到堆中，然后使用 remove 方法从堆中删除所有元素。以降序删除这些元素。程序清单 23-10 给出使用堆对数组排序的算法。

程序清单 23-10 HeapSort.java

```

1 public class HeapSort {
2     /** Heap sort method */
3     public static <E extends Comparable<E>> void heapSort(E[] list) {
4         // Create a Heap of integers
5         Heap<E> heap = new Heap<>();
6
7         // Add elements to the heap
8         for (int i = 0; i < list.length; i++)
9             heap.add(list[i]);
10
11         // Remove elements from the heap
12         for (int i = list.length - 1; i >= 0; i--)

```

```

13     list[i] = heap.remove();
14 }
15
16 /** A test method */
17 public static void main(String[] args) {
18     Integer[] list = {-44, -5, -3, 3, 3, 1, -4, 0, 1, 2, 4, 5, 53};
19     heapSort(list);
20     for (int i = 0; i < list.length; i++)
21         System.out.print(list[i] + " ");
22 }
23 }

```

-44 -5 -4 -3 0 1 1 2 3 3 4 5 53

23.6.6 堆排序的时间复杂度

下面将注意力转到分析堆排序的时间复杂度上。设 h 表示包含 n 个元素的堆的高度。由于堆是一棵完全二叉树，所以，第一层有 1 个结点，第二层有 2 个结点，第 k 层有 2^{k-1} 个结点，第 $h-1$ 层有 2^{h-2} 个结点，而第 h 层最少有一个结点且最多有 2^{h-1} 个结点。因此

$$1 + 2 + \cdots + 2^{h-2} < n \leq 1 + 2 + \cdots + 2^{h-2} + 2^{h-1}$$

也就是

$$2^{h-1} - 1 < n \leq 2^h - 1$$

$$2^{h-1} < n+1 \leq 2^h$$

$$h-1 < \log(n+1) \leq h$$

这样， $h < \log(n+1) + 1$ 和 $\log(n+1) \leq h$ 。因此， $\log(n+1) \leq h < \log(n+1) + 1$ 。所以，堆的高度为 $O(\log n)$ 。

由于 `add` 方法会追踪从叶子结点到根结点的路径，因此向堆中添加一个新元素最多需要 h 步。所以，建立一个包含 n 个元素的数组的初始堆需要 $O(n \log n)$ 时间。因为 `remove` 方法要追踪从根结点到叶子结点的路径，因此从堆中删除根结点后，重建堆最多需要 h 步。由于要调用 n 次 `remove` 方法，所以由堆产生一个有序数组需要的总时间为 $O(n \log n)$ 。

归并排序和堆排序需要的时间都为 $O(n \log n)$ 。为归并两个子数组，归并排序需要一个临时数组，而堆排序不需要额外的数组空间。因此，堆排序的空间效率高于归并排序。

✓ 复习题

23.13 什么是完全二叉树？什么是堆？描述如何从堆中删除根结点，以及如何向堆中增加一个对象。

23.14 如果堆为空，那么调用 `remove` 方法的返回值是什么？

23.15 顺序添加元素 4, 5, 1, 2, 9 和 3 到一个堆中，画一个图来演示添加每个元素后堆的情况。

23.16 绘制一幅图，显示图 23-15c 中堆的根结点被删除后的堆。

23.17 插入一个新元素到一个堆中的时间复杂度是多少？从一个堆中删除一个元素的时间复杂度是多少？

23.18 显示使用 {45, 11, 50, 59, 60, 2, 4, 7, 10} 创建一个堆的步骤。

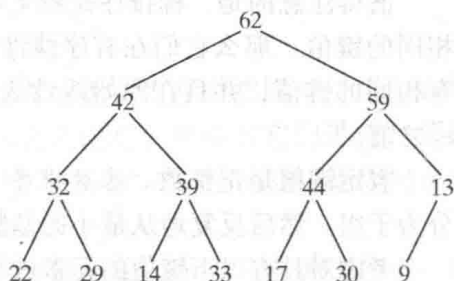
23.19 给定下面的堆，描述从堆中删除所有结点的步骤。

23.20 下面的语句哪个是错误的？

```

1 Heap<Object> heap1 = new Heap<>();
2 Heap<Number> heap2 = new Heap<>();
3 Heap<BigInteger> heap3 = new Heap<>();
4 Heap<Calendar> heap4 = new Heap<>();
5 Heap<String> heap5 = new Heap<>();

```

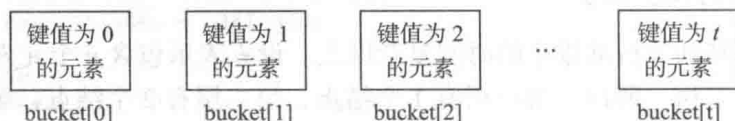


23.7 桶排序和基数排序

🔑 要点提示：桶排序和基数排序是对整数进行排序的高效算法。

目前所讨论的所有排序算法都是可以用在任何键值类型（例如，整数、字符串以及任何可比较的对象）上的通用排序算法。这些算法都是通过比较它们的键值来对元素排序的。已经证明，基于比较的排序算法的复杂度不会好于 $O(n\log n)$ 。但是，如果键值是整数，那么可以使用桶排序，而无须比较这些键值。

桶排序算法的工作方式如下。假设键值的范围是从 0 到 t 。我们需要 $t+1$ 个标记为 0, 1, ..., t 的桶。如果元素的键值是 i ，那么就将该元素放入桶 i 中。每个桶中都放着具有相同键值的元素。



可以使用 `ArrayList` 来实现一个桶。应用桶排序算法对一个元素线性表进行排序的过程可以描述如下：

```
void bucketSort(E[] list) {
    E[] bucket = (E[])new java.util.ArrayList[t+1];

    // Distribute the elements from list to buckets
    for (int i = 0; i < list.length; i++) {
        int key = list[i].getKey(); // Assume element has the getKey() method
        if (bucket[key] == null)
            bucket[key] = new java.util.ArrayList<>();
        bucket[key].add(list[i]);
    }

    // Now move the elements from the buckets back to list
    int k = 0; // k is an index for list
    for (int i = 0; i < bucket.length; i++) {
        if (bucket[i] != null) {
            for (int j = 0; j < bucket[i].size(); j++)
                list[k++] = bucket[i].get(j);
        }
    }
}
```

很明显，它需要耗费 $O(n+t)$ 时间来对线性表排序，使用的空间是 $O(n+t)$ ，其中 n 是指线性表的大小。

注意，如果 t 太大，那么桶排序不是很可取。此时，可以使用基数排序。基数排序是基于桶排序的，但是它只使用 10 个桶。

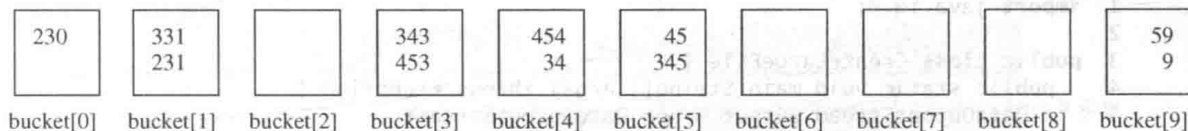
值得注意的是，桶排序是稳定的（stable），这意味着，如果原始线性表中的两个元素有相同的键值，那么它们在有序线性表中的顺序是不变的。也就是说，如果元素 e_1 和元素 e_2 有相同的键值，并且在原始线性表中， e_1 在 e_2 之前，那么在排好序的线性表中， e_1 还是在 e_2 之前。

假定键值是正整数。基数排序（radix sort）的思路就是将这些键值基于它们的基数位置分为子组。然后反复地从最小的基数位置开始，对其上的键值应用桶排序。

考虑对具有以下键值的元素进行排序：

331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9

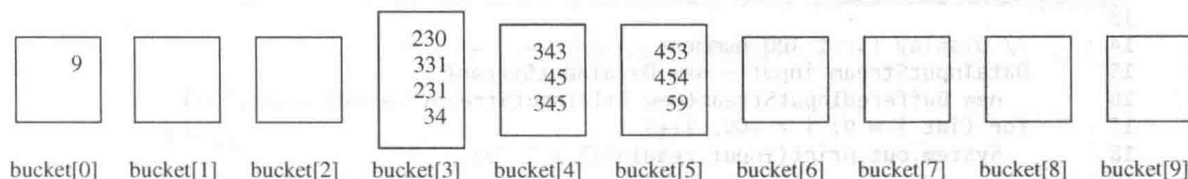
在最后一位基数位置上应用桶排序。这些元素被按如下方式放在桶中：



将元素从桶中删除之后，它们以下面的顺序排列：

230, 331, 231, 343, 453, 454, 34, 45, 345, 59, 9

在倒数第二位基数位置上应用桶排序。这些元素被按如下方式放在桶中：

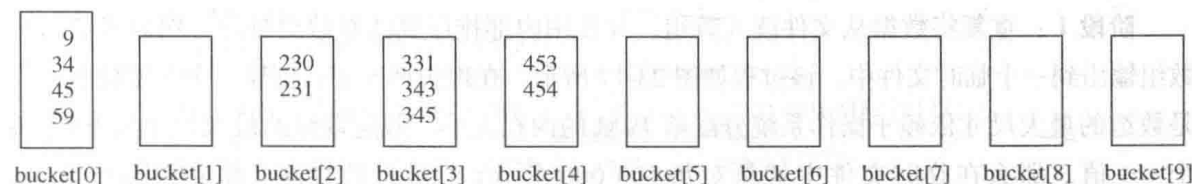


将元素从桶中删除之后，它们以下面的顺序排列：

9, 230, 331, 231, 34, 343, 45, 345, 453, 454, 59

(注意，9 是 009。)

在倒数第三位基数位置上应用桶排序。这些元素被按如下方式放在桶中：



将元素从桶中删除之后，它们以下面的顺序排列：

9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454

现在这些元素是有序的了。

通常，基数排序需要耗费 $O(dn)$ 时间对带整数键值的 n 个元素排序，其中 d 是所有键值中基数位数的最大值。

复习题

23.21 可以使用桶排序来对一个字符串线性表进行排序吗？

23.22 使用数字 454, 34, 23, 43, 74, 86 以及 76 来演示基数排序是如何进行排序的。

23.8 外部排序

要点提示：可以使用外部排序来对大容量数据进行排序。

前面几节讨论的所有排序算法，都假定要排序的所有数据在内存中都同时可用，如数组。要对存储在外部文件中的数据排序，首先要将数据送入内存，然后对它们进行内部排序。然而，如果文件太大，那么文件中的所有数据不能同时送入内存。本节将讨论如何在大型外部文件中对数据排序。这称为外部排序 (external sort)。

为简单起见，假定将 200 万个 int 值存储在一个名为 largedata.dat 的二进制文件中。该

文件是使用程序清单 23-11 中的程序创建的。

程序清单 23-11 CreateLargeFile.java

```

1  import java.io.*;
2
3  public class CreateLargeFile {
4      public static void main(String[] args) throws Exception {
5          DataOutputStream output = new DataOutputStream(
6              new BufferedOutputStream(
7                  new FileOutputStream("largedata.dat")));
8
9          for (int i = 0; i < 800004; i++)
10             output.writeInt((int)(Math.random() * 1000000));
11
12         output.close();
13
14         // Display first 100 numbers
15         DataInputStream input = new DataInputStream(
16             new BufferedInputStream(new FileInputStream("largedata.dat")));
17         for (int i = 0; i < 100; i++)
18             System.out.print(input.readInt() + " ");
19
20         input.close();
21     }
22 }

```

569193 131317 608695 776266 767910 624915 458599 5010 ... (omitted)

可以使用归并排序的一种变体对这个文件进行两步排序：

阶段 I：重复将数据从文件读入数组，并使用内部排序算法对数组排序，然后将数据从数组输出到一个临时文件中。该过程如图 23-17 所示。在理想情况下，创建一个大型数组，但是数组的最大尺寸依赖于操作系统分配给 JVM 的内存大小。假定数组的最大尺寸为 100 000 个 int 值，那么在临时文件中就是对每 100 000 个 int 值进行的排序。将它们标记为 S_1, S_2, \dots, S_k 其中，最后一段 S_k ，包含的数值可能会少于 100 000 个。

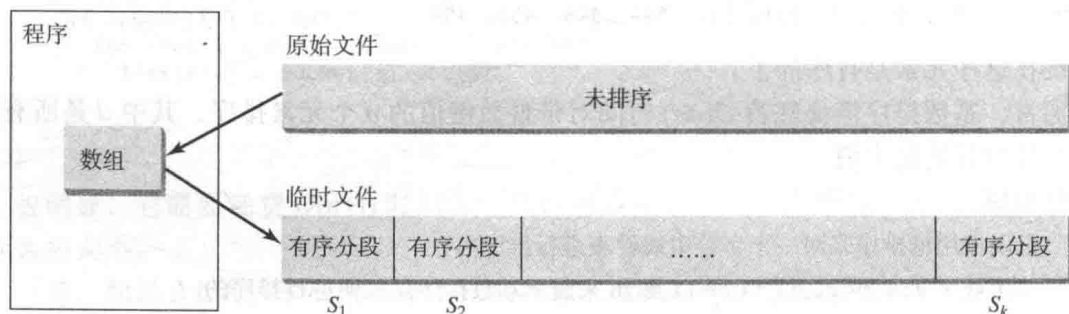


图 23-17 对原始文件分段排序

阶段 II：将每对有序分段（比如 S_1 和 S_2 , S_3 和 S_4 , ...）归并到一个大一些的有序分段中，并将新分段存储到新的临时文件中。继续同样的过程直到得到仅仅一个有序分段。图 23-18 演示了如何对 8 个分段进行归并。

注意：不一定要归并两个相邻分段。例如，在第一步归并中，可以归并 S_1 和 S_5 、 S_2 和 S_6 、 S_3 和 S_7 、 S_4 和 S_8 。这在高效实现阶段 II 时很有用。

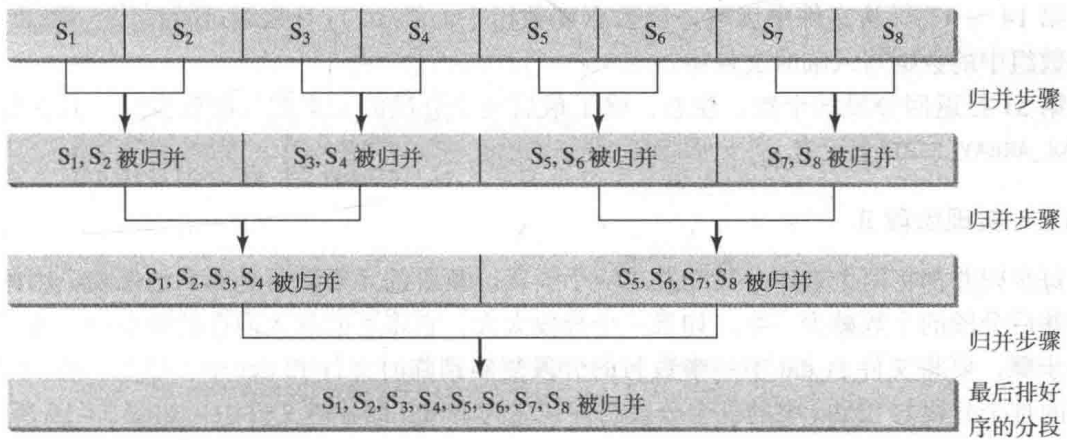


图 23-18 对有序分段进行迭代归并

23.8.1 实现阶段 I

程序清单 23-12 给出一个方法，它从文件中读取每个数据段，并对分段进行排序，然后将排好序的分段存在一个新文件中。该方法返回分段的个数。

程序清单 23-12 创建初始的有序分段

```

1  /** Sort original file into sorted segments */
2  private static int initializeSegments
3      (int segmentSize, String originalFile, String f1)
4      throws Exception {
5      int[] list = new int[segmentSize];
6      DataInputStream input = new DataInputStream(
7          new BufferedInputStream(new FileInputStream(originalFile)));
8      DataOutputStream output = new DataOutputStream(
9          new BufferedOutputStream(new FileOutputStream(f1)));
10
11     int numberOfSegments = 0;
12     while (input.available() > 0) {
13         numberOfSegments++;
14         int i = 0;
15         for (; input.available() > 0 && i < segmentSize; i++) {
16             list[i] = input.readInt();
17         }
18
19         // Sort an array list[0..i-1]
20         java.util.Arrays.sort(list, 0, i);
21
22         // Write the array to f1.dat
23         for (int j = 0; j < i; j++) {
24             output.writeInt(list[j]);
25         }
26     }
27
28     input.close();
29     output.close();
30
31     return numberOfSegments;
32 }

```

该方法在第 5 行创建一个具有最大尺寸的数组，在第 6 行为原始文件创建一个数据输入流，在第 8 行为临时文件创建一个数据输出流。缓冲流用于提高程序性能。

第 14 ~ 17 行从文件中读取一段数据到数组中。第 20 行对数组进行排序。第 23 ~ 25 行将数组中的数据写入临时文件中。

第 31 行返回分段的个数。注意，除了最后一个分段的元素数可能较少外，其他分段都有 MAX_ARRAY_SIZE 个元素。

23.8.2 实现阶段 II

每步归并都将两个有序分段归并成一个新段。新段的元素数目是原来的两倍，因此，每次归并后分段的个数减少一半。如果一个分段太大，它将不能放入内存的数组中。为了实现归并步骤，要将文件 f1.dat 中一半数目的分段复制到临时文件 f2.dat 中。然后，将 f1.dat 中剩下的首个分段与 f2.dat 中的首个分段归并到名为 f3.dat 的临时文件中，如图 23-19 所示。

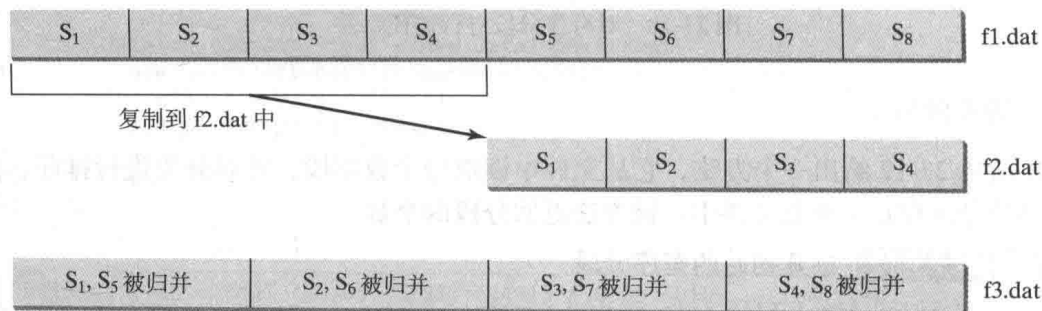


图 23-19 迭代归并有序分段

注意：f1.dat 可能会比 f2.dat 多一个分段。这样的话，在归并后将最后一个分段移到 f3.dat 中。

程序清单 23-13 给出一个方法，将 f1.dat 中的前半部分复制到 f2.dat 中。程序清单 23-14 给出一个方法，将 f1.dat 和 f2.dat 中的一对分段进行归并。程序清单 23-15 给出一个方法，对两个分段进行归并。

程序清单 23-13 复制前半部分的分段

```
1 private static void copyHalfToF2(int numberOfSegments,
2     int segmentSize, DataInputStream f1, DataOutputStream f2)
3     throws Exception {
4     for (int i = 0; i < (numberOfSegments / 2) * segmentSize; i++) {
5         f2.writeInt(f1.readInt());
6     }
7 }
```

程序清单 23-14 归并所有分段

```
1 private static void mergeSegments(int numberOfSegments,
2     int segmentSize, DataInputStream f1, DataInputStream f2,
3     DataOutputStream f3) throws Exception {
4     for (int i = 0; i < numberOfSegments; i++) {
5         mergeTwoSegments(segmentSize, f1, f2, f3);
6     }
7
8     // If f1 has one extra segment, copy it to f3
9     while (f1.available() > 0) {
10         f3.writeInt(f1.readInt());
11     }
12 }
```

程序清单 23-15 归并两个分段

```

1 private static void mergeTwoSegments(int segmentSize,
2   DataInputStream f1, DataInputStream f2,
3   DataOutputStream f3) throws Exception {
4   int intFromF1 = f1.readInt();
5   int intFromF2 = f2.readInt();
6   int f1Count = 1;
7   int f2Count = 1;
8
9   while (true) {
10    if (intFromF1 < intFromF2) {
11      f3.writeInt(intFromF1);
12      if (f1.available() == 0 || f1Count++ >= segmentSize) {
13        f3.writeInt(intFromF2);
14        break;
15      }
16    } else {
17      intFromF1 = f1.readInt();
18    }
19  }
20  else {
21    f3.writeInt(intFromF2);
22    if (f2.available() == 0 || f2Count++ >= segmentSize) {
23      f3.writeInt(intFromF1);
24      break;
25    }
26  } else {
27    intFromF2 = f2.readInt();
28  }
29 }
30 }
31
32 while (f1.available() > 0 && f1Count++ < segmentSize) {
33   f3.writeInt(f1.readInt());
34 }
35
36 while (f2.available() > 0 && f2Count++ < segmentSize) {
37   f3.writeInt(f2.readInt());
38 }
39 }

```

23.8.3 结合两个阶段

程序清单 23-16 给出一个完整的程序，对 largedata.dat 中的 int 值进行排序，并将已排好序的数据存储在 sortedfile.dat 中。

程序清单 23-16 SortLargeFile.java

```

1 import java.io.*;
2
3 public class SortLargeFile {
4   public static final int MAX_ARRAY_SIZE = 100000;
5   public static final int BUFFER_SIZE = 100000;
6
7   public static void main(String[] args) throws Exception {
8     // Sort largedata.dat to sortedfile.dat
9     sort("largedata.dat", "sortedfile.dat");
10
11     // Display the first 100 numbers in the sorted file
12     displayFile("sortedfile.dat");
13   }
14 }

```

```

15  /** Sort data in source file into target file */
16  public static void sort(String sourcefile, String targetfile)
17      throws Exception {
18      // Implement Phase 1: Create initial segments
19      int numberOfSegments =
20          initializeSegments(MAX_ARRAY_SIZE, sourcefile, "f1.dat");
21
22      // Implement Phase 2: Merge segments recursively
23      merge(numberOfSegments, MAX_ARRAY_SIZE,
24          "f1.dat", "f2.dat", "f3.dat", targetfile);
25  }
26
27  /** Sort original file into sorted segments */
28  private static int initializeSegments
29      (int segmentSize, String originalFile, String f1)
30      throws Exception {
31      // Same as Listing 23.12, so omitted
32  }
33
34  private static void merge(int numberOfSegments, int segmentSize,
35      String f1, String f2, String f3, String targetfile)
36      throws Exception {
37      if (numberOfSegments > 1) {
38          mergeOneStep(numberOfSegments, segmentSize, f1, f2, f3);
39          merge((numberOfSegments + 1) / 2, segmentSize * 2,
40              f3, f1, f2, targetfile);
41      }
42      else { // Rename f1 as the final sorted file
43          File sortedFile = new File(targetfile);
44          if (sortedFile.exists()) sortedFile.delete();
45          new File(f1).renameTo(sortedFile);
46      }
47  }
48
49  private static void mergeOneStep(int numberOfSegments,
50      int segmentSize, String f1, String f2, String f3)
51      throws Exception {
52      DataInputStream f1Input = new DataInputStream(
53          new BufferedInputStream(new FileInputStream(f1), BUFFER_SIZE));
54      DataOutputStream f2Output = new DataOutputStream(
55          new BufferedOutputStream(new FileOutputStream(f2), BUFFER_SIZE));
56
57      // Copy half number of segments from f1.dat to f2.dat
58      copyHalfToF2(numberOfSegments, segmentSize, f1Input, f2Output);
59      f2Output.close();
60
61      // Merge remaining segments in f1 with segments in f2 into f3
62      DataInputStream f2Input = new DataInputStream(
63          new BufferedInputStream(new FileInputStream(f2), BUFFER_SIZE));
64      DataOutputStream f3Output = new DataOutputStream(
65          new BufferedOutputStream(new FileOutputStream(f3), BUFFER_SIZE));
66
67      mergeSegments(numberOfSegments / 2,
68          segmentSize, f1Input, f2Input, f3Output);
69
70      f1Input.close();
71      f2Input.close();
72      f3Output.close();
73  }
74
75  /** Copy first half number of segments from f1.dat to f2.dat */
76  private static void copyHalfToF2(int numberOfSegments,
77      int segmentSize, DataInputStream f1, DataOutputStream f2)

```

```

78     throws Exception {
79     // Same as Listing 23.13, so omitted
80 }
81
82 /** Merge all segments */
83 private static void mergeSegments(int numberOfSegments,
84     int segmentSize, DataInputStream f1, DataInputStream f2,
85     DataOutputStream f3) throws Exception {
86     // Same as Listing 23.14, so omitted
87 }
88
89 /** Merges two segments */
90 private static void mergeTwoSegments(int segmentSize,
91     DataInputStream f1, DataInputStream f2,
92     DataOutputStream f3) throws Exception {
93     // Same as Listing 23.15, so omitted
94 }
95
96 /** Display the first 100 numbers in the specified file */
97 public static void displayFile(String filename) {
98     try {
99         DataInputStream input =
100             new DataInputStream(new FileInputStream(filename));
101         for (int i = 0; i < 100; i++)
102             System.out.print(input.readInt() + " ");
103         input.close();
104     }
105     catch (IOException ex) {
106         ex.printStackTrace();
107     }
108 }
109 }

```

0 1 1 1 2 2 2 3 3 4 5 6 8 8 9 9 9 10 10 11 . . . (omitted)
--

在运行该程序之前，首先运行程序清单 23-11 来创建 `largedata.dat`。调用 `sort("largedata.dat", "sortedfile.dat")` (第 9 行) 从 `largedata.dat` 中读取数据并向 `sortedfile.dat` 写入排好序的数据。调用 `displayFile("sortedfile.dat")` (第 12 行) 显示特定文件中的前 100 个数字。注意，这个文件是用二进制 I/O 创建的，因而不能使用文本编辑器 (如记事本) 来查看它。

`sort` 方法首先从原始数组中创建初始分段，并且将排好序的分段存入新文件 `f1.dat` 中 (第 19 ~ 20 行)，然后在 `targetfile` 中就产生了一个有序文件 (第 23 ~ 24 行)。

merge 方法

```
merge(int numberOfSegments, int segmentSize,
      String f1, String f2, String f3, String targetfile)
```

使用 `f2` 作为辅助将 `f1` 中的分段归并到 `f3` 中。`merge` 方法在很多归并步骤中都会被递归调用。每步归并都会使分段数 `numberOfSegments` 减少一半，同时使有序分段规模翻倍。在完成一个归并步骤后，下一个归并步骤使用 `f1` 作为辅助将 `f3` 中的新分段归并到 `f2` 中。因此，调用新归并方法的语句为

```
merge((numberOfSegments + 1) / 2, segmentSize * 2,
      f3, f1, f2, targetfile);
```

下一个归并步骤的 `numberOfSegments` 为 $(\text{numberOfSegments} + 1) / 2$ 。例如，如果 `numberOfSegments` 为 5，那么，下一个归并步骤的 `numberOfSegments` 为 3，因为每两个分段进行归

并时会留下一个未归并的分段。

当 `numberOfSegments` 为 1 时, 结束递归的 `merge` 方法。在这种情况下, `f1` 中包含已排好序的数据。文件 `f1` 被重命名为 `targetfile` (第 45 行)。

23.8.4 外部排序复杂度

在外部排序中, 主要开销是在 I/O 上。假设 n 是文件中要排序的元素个数。在阶段 I, 从原始文件中读取元素个数 n , 然后将它输出给一个临时文件。因此, 阶段 I 的 I/O 复杂度为 $O(n)$ 。

对于阶段 II, 在第一个合并步骤之前, 排好序的分段的个数为 $\frac{n}{c}$, 其中 c 是 `MAX_ARRAY_SIZE`。每一个合并步骤都会使分段的个数减半。因此, 在第一次合并步骤之后, 分段个数为 $\frac{n}{2c}$ 。在第二次合并步骤之后, 分段个数为 $\frac{n}{2^2c}$ 。在第三次合并步骤之后, 分段个数为 $\frac{n}{2^3c}$ 。在第 $\log\left(\frac{n}{c}\right)$ 次合并步骤之后, 分段个数减到 1。因此, 合并步骤的总数为 $\log\left(\frac{n}{c}\right)$ 。

在每次合并步骤中, 从文件 `f1` 读取一半数量的分段, 然后将它们写入一个临时文件 `f2`。合并 `f1` 中剩余的分段和 `f2` 中的分段。每一个合并步骤中 I/O 的次数为 $O(n)$ 。因为合并步骤的总数是 $\log\left(\frac{n}{c}\right)$, I/O 的总数是

$$O(n) \times \log\left(\frac{n}{c}\right) = O(n \log n)$$

因此, 外部排序的复杂度是 $O(n \log n)$ 。

✓ 复习题

23.23 描述外部排序是如何工作的。外部排序算法的复杂度是多少?

23.24 10 个数字 {2,3,4,0,5,6,7,9,8,1} 保存在外部文件 `largedata.dat` 中。设 `MAX_ARRAY_SIZE` 为 2, 手工跟踪 `SortLargeFile` 程序。

关键术语

bubble sort (冒泡排序)

bucket sort (桶排序)

complete binary tree (完全二叉树)

external sort (外部排序)

heap (堆)

heap sort (堆排序)

height of a heap (堆的高度)

merge sort (归并排序)

quick sort (快速排序)

radix sort (基数排序)

本章小结

1. 选择排序、插入排序、冒泡排序和快速排序的最差时间复杂度为 $O(n^2)$ 。
2. 归并排序的平均情况和最差情况的复杂度为 $O(n \log n)$ 。快速排序的平均时间也是 $O(n \log n)$ 。
3. 对于设计排序这样的高效算法, 堆是一个很有用的数据结构。本章介绍了如何定义和实现一个堆类, 以及如何向 / 从堆中插入和删除元素。
4. 堆排序的时间复杂度为 $O(n \log n)$ 。
5. 桶排序和基数排序都是针对整数键值的特定排序算法。这些算法不是通过比较键值而是使用桶来对键值排序的, 它们会比一般的排序算法效率更高。

6. 可以使用归并排序的一种变体——称为外部排序——对外部文件中的大型数据进行排序。

测试题

回答位于网址 www.cs.armstrong.edu/liang/intro10e/test.html 的本章测试题。

编程练习题

23.3 ~ 23.5 节

23.1 (泛型冒泡排序) 使用冒泡排序编写下面两个泛型方法。第一个方法使用 `Comparable` 接口对元素排序, 第二个方法使用 `Comparator` 接口对元素排序。

```
public static <E extends Comparable<E>>
    void bubbleSort(E[] list)
public static <E> void bubbleSort(E[] list,
    Comparator<? super E> comparator)
```

23.2 (泛型归并排序) 使用归并排序编写下面两个泛型方法。第一个方法使用 `Comparable` 接口对元素排序, 第二个方法使用 `Comparator` 接口对元素排序。

```
public static <E extends Comparable<E>>
    void mergeSort(E[] list)
public static <E> void mergeSort(E[] list,
    Comparator<? super E> comparator)
```

23.3 (泛型快速排序) 使用快速排序编写下面两个泛型方法。第一个方法使用 `Comparable` 接口对元素排序, 第二个方法使用 `Comparator` 接口对元素排序。

```
public static <E extends Comparable<E>>
    void quickSort(E[] list)
public static <E> void quickSort(E[] list,
    Comparator<? super E> comparator)
```

23.4 (改进快速排序) 本书提供的快速排序算法选择线性表中的第一个元素作为主元。修改该算法, 在线性表中的第一个元素、中间元素和最后一个元素中选择一个中位数作为主元。

*23.5 (泛型堆排序) 使用堆排序编写下面两个泛型方法。第一个方法使用 `Comparable` 接口对元素排序, 第二个方法使用 `Comparator` 接口对元素排序。

```
public static <E extends Comparable<E>>
    void heapSort(E[] list)
public static <E> void heapSort(E[] list,
    Comparator<? super E> comparator)
```

23.6 (检查顺序) 编写下面的重载方法, 用于检查数组是按升序还是降序排列的。默认情况下, 该方法是检查升序的。为检查降序, 则将 `false` 传递给方法中的升序参数。

```
public static boolean ordered(int[] list)
public static boolean ordered(int[] list, boolean ascending)
public static boolean ordered(double[] list)
public static boolean ordered
    (double[] list, boolean ascending)
public static <E extends Comparable<E>>
    boolean ordered(E[] list)
public static <E extends Comparable<E>> boolean ordered
    (E[] list, boolean ascending)
public static <E> boolean ordered(E[] list,
    Comparator<? super E> comparator)
public static <E> boolean ordered(E[] list,
    Comparator<? super E> comparator, boolean ascending)
```

23.6 节

23.7 (最小堆) 本书中介绍的堆也称为最大堆 (max-heap), 其中的每个结点都大于或等于它的任何一

个子结点。最小堆 (min-heap) 是指每个结点都小于或等于它的任何一个子结点的堆。修改程序清单 23-9 中的 Heap 类以实现最小堆。

*23.8 (使用堆排序) 使用堆实现下面的 sort 方法。

```
public static <E extends Comparable<E>> void sort(E[] list)
```

*23.9 (使用 Comparator 的泛型堆) 修改程序清单 23-9 中的 Heap, 使用泛型参数和一个 Comparator 来比较对象。定义一个新的构造方法, 以 Comparator 作为它的参数, 如下所示:

```
Heap(Comparator<? super E> comparator)
```

**23.10 (堆的可视化) 编写一个程序, 图形化显示一个堆, 如图 23-10 所示。该程序允许用户向堆中插入和从堆中删除元素。

23.11 (堆的 clone 和 equals 方法) 实现 Heap 类中的 clone 和 equals 方法。

23.7 节

*23.12 (基数排序) 编写程序, 随机创建 1 000 000 个整数, 然后使用基数排序对它们排序。

*23.13 (排序的执行时间) 编写程序, 获取输入规模为 50 000、100 000、150 000、200 000、250 000 和 300 000 时的选择排序、冒泡排序、归并排序、快速排序、堆排序以及基数排序的执行时间。该程序应随机地创建数据, 然后打印如下所示的一个表格:

数组大小	选择排序	冒泡排序	归并排序	快速排序	堆排序	基数排序
50 000						
100 000						
150 000						
200 000						
250 000						
300 000						

(提示: 可以使用下面的代码模板来获取执行时间。)

```
long startTime = System.currentTimeMillis();
perform the task;
long endTime = System.currentTimeMillis();
long executionTime = endTime - startTime;
```

本书给出了一个递归的快速排序, 在此编写一个非递归版本。

23.8 节

*23.14 (外部排序的执行时间) 编写程序, 获取输入规模为 5 000 000、10 000 000、15 000 000、20 000 000、25 000 000 和 30 000 000 时外部排序的执行时间。该程序应该打印出如下所示的一个表格:

文件尺寸	5 000 000	10 000 000	15 000 000	20 000 000	25 000 000	30 000 000
时间						

综合

*23.15 (选择排序动画) 编写一个程序, 实现选择排序算法的动画。创建一个数组, 以随机顺序包含从 1 到 20 的 20 个不同数字。数组元素在一个直方图中显示, 如图 23-20a 所示。单击 Step 按钮使程序执行算法中外部循环的一次迭代, 然后为新的数组重画直方图。将排好序的子数组标上颜色。当算法结束时, 显示一条信息通知用户。单击 Reset 按钮为一次新的开始创建一个新的随机数组。(可以很容易地修改程序, 来制作插入排序算法的动画。)

*23.16 (冒泡排序动画) 编写一个程序, 实现冒泡排序算法的动画。创建一个数组, 以随机顺序包含从

1 到 20 的 20 个不同数字。数组元素在一个直方图中显示, 如图 23-20b 所示。单击 Step 按钮使程序执行算法中的一次比较, 然后为新的数组重画直方图。将表示考虑交换的数值条标上颜色。当算法结束时, 显示一条信息通知用户。单击 Reset 按钮为一次新的开始创建一个新的随机数组。

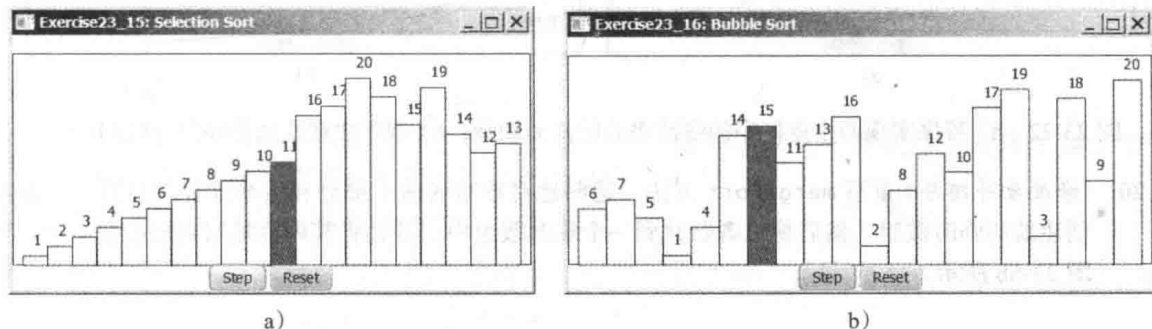


图 23-20 a) 程序实现选择排序的动画; b) 程序实现冒泡排序的动画

- *23.17 (基数排序动画) 编写一个程序, 实现基数排序算法的动画。创建一个数组, 以随机顺序包含从 1 到 1000 的 20 个不同数字。数组元素在一个直方图中显示, 如图 23-21 所示。单击 Step 按钮使程序放置一个数字在一个桶中。刚放入的数字以红色显示。一旦所有的数字都放在桶中后, 单击 Step 按钮从桶中收集所有的数字, 将它们移回到数组中。当算法结束时, 单击 Step 按钮显示一条信息通知用户。单击 Reset 按钮为一次新的开始创建一个新的随机数组。

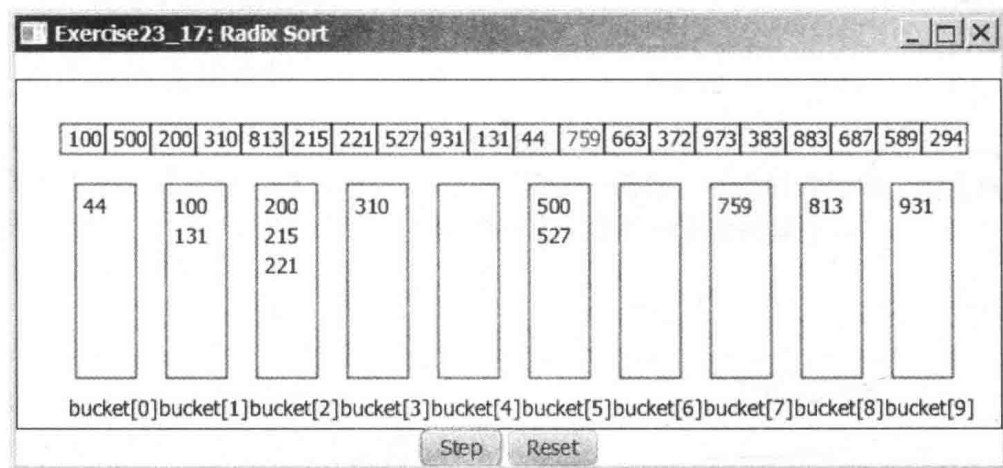


图 23-21 程序实现基数排序的动画

- *23.18 (归并排序动画) 编写一个程序, 实现两个排好序的线性表的归并的动画。创建两个数组, list1 和 list2, 每个包含从 1 到 999 的 8 个随机数字。数组元素如图 23-22a 所示。单击 Step 按钮使程序将 list1 或者 list2 中的一个元素移到 temp 中。单击 Reset 按钮为一个新的开始创建两个新的随机数组。当算法结束时, 单击 Step 按钮显示一条信息通知用户。
- *23.19 (快速排序分区动画) 编写一个程序, 实现快速排序的分区动画。程序创建一个包含从 1 到 999 的 20 个随机数字的线性表。线性表如图 23-22b 所示。单击 Step 按钮使程序将 low 移动到右边, 或者 high 移动到左边, 或者交换 low 和 high 位置的元素。单击 Reset 按钮为一个新的开始创建两个新的随机数组。当算法结束时, 单击 Step 按钮显示一条信息通知用户。

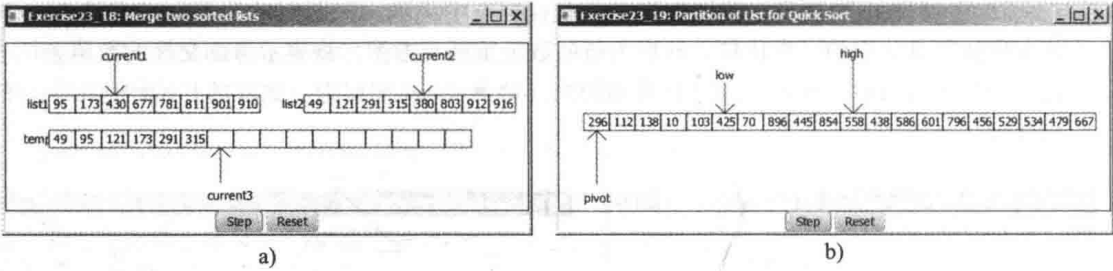


图 23-22 a) 程序实现两个排好序的线性表的归并的动画；b) 程序实现快速排序的分区动画

*23.20 (修改合并排序) 重写 `mergeSort` 方法，递归地对数组的前半部分和后半部分进行排序，而不创建新的临时数组。然后将二者归并到一个临时数组中，并且将其内容复制到原始数组中，如图 23-6b 所示。

实现线性表、栈、队列和优先队列

{} 教学目标

- 在接口中设计线性表的通用特性，并且在一个便利抽象类中提供骨架实现（24.2 节）。
- 使用数组设计并实现数组线性表（24.3 节）。
- 使用链式结构设计并实现链表（24.4 节）。
- 使用数组线性表设计并实现一个栈类，使用链表实现一个队列类（24.5 节）。
- 使用堆设计并实现优先队列（24.6 节）。

24.1 引言

🔑 要点提示：本章专注于实现数据结构。

线性表、栈、队列和优先队列都是典型的数据结构，经常会在数据结构课程中讲到。Java API 中对它们有支持，关于它们的使用方法已在第 20 章中给出。本章将剖析这些数据结构是如何实现的。集合和映射表的实现将在第 27 章中讲述。通过这些例子，你将学到如何设计和实现自定义的数据结构。

24.2 线性表的通用特性

🔑 要点提示：线性表的通用特性在 List 接口中定义。

线性表是一个顺序存储数据的流行数据结构——例如，学生的线性表、空房间的线性表、城市的线性表以及书籍的线性表。可以在线性表上执行下面的操作：

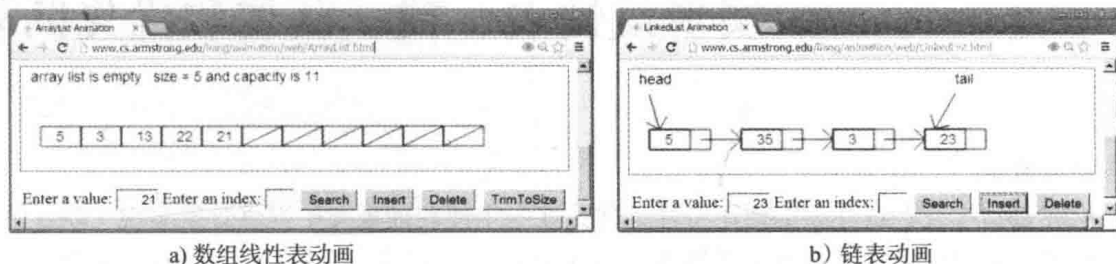
- 从线性表中提取一个元素。
- 向线性表中插入一个新元素。
- 从线性表中删除一个元素。
- 找出线性表中元素的个数。
- 确定线性表中是否包含某个元素。
- 确定线性表是否为空。

实现线性表的方式有两种。一种是使用数组（array）存储线性表的元素。数组大小是固定的。如果元素个数超过了数组的容量，就创建一个更大的新数组，并将当前数组中的元素复制到新数组中。另一种是使用链式结构（linked structure）。链式结构由结点组成，每个结点都是动态创建的，用来存储一个元素。所有的结点链接成一个线性表。这样，就可以给线性表定义两种类。为了方便起见，分别称这两种类为 MyArrayList 和 MyLinkedList。这两种类具有相同的操作，但是具有不同的实现。

{} 设计指南：通用的操作可以归纳为一个接口或者一个抽象类。一个好的策略就是在设计中提供接口和便利抽象类，以整合接口和抽象类的优点，这样用户可以认为哪个方便就用哪个。抽象类提供了接口的骨架实现，可以更有效地实现接口。

{} 教学注意：参见链接 www.cs.armstrong.edu/liang/animation/web/ArrayList.html 和 www.cs.armstrong.edu/liang/animation/web/LinkedList.html。

cs.armstrong.edu/liang/animation/web/LinkedList.html 来查看数组线性表和链表的在线交互式演示, 如图 24-1 所示。



a) 数组线性表动画

b) 链表动画

图 24-1 动画工具有助于了解数组线性表和链表是如何工作的

我们把这样的接口命名为 `MyList`, 把便利类命名为 `MyAbstractList`。图 24-2 展示了 `MyList`、`MyAbstractList`、`MyArrayList` 以及 `MyLinkedList` 之间的关系。图 24-3 列出了 `MyList` 中的方法和 `MyAbstractList` 中实现的方法。程序清单 24-1 给出 `MyList` 的源代码。

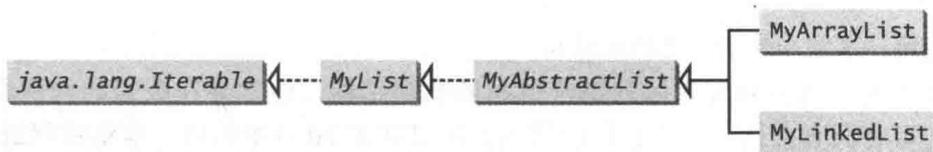


图 24-2 `MyList` 定义了 `MyAbstractList`、`MyArrayList` 和 `MyLinkedList` 的通用接口

程序清单 24-1 `MyList.java`

```

1 public interface MyList<E> extends java.lang.Iterable<E> {
2     /** Add a new element at the end of this list */
3     public void add(E e);
4
5     /** Add a new element at the specified index in this list */
6     public void add(int index, E e);
7
8     /** Clear the list */
9     public void clear();
10
11     /** Return true if this list contains the element */
12     public boolean contains(E e);
13
14     /** Return the element from this list at the specified index */
15     public E get(int index);
16
17     /** Return the index of the first matching element in this list.
18      * Return -1 if no match. */
19     public int indexOf(E e);
20
21     /** Return true if this list doesn't contain any elements */
22     public boolean isEmpty();
23
24     /** Return the index of the last matching element in this list
25      * Return -1 if no match. */
26     public int lastIndexOf(E e);
27
28     /** Remove the first occurrence of the element e from this list.
29      * Shift any subsequent elements to the left.
30      * Return true if the element is removed. */
31     public boolean remove(E e);
32
33     /** Remove the element at the specified position in this list.
  
```

```

34  * Shift any subsequent elements to the left.
35  * Return the element that was removed from the list. */
36  public E remove(int index);
37
38  /** Replace the element at the specified position in this list
39  * with the specified element and return the old element. */
40  public Object set(int index, E e);
41
42  /** Return the number of elements in this list */
43  public int size();
44  }

```

MyAbstractList 声明变量 size，表示线性表中元素的个数。程序清单 24-2 中的类可以实现 isEmpty()、size()、add(E) 和 remove(E) 方法。

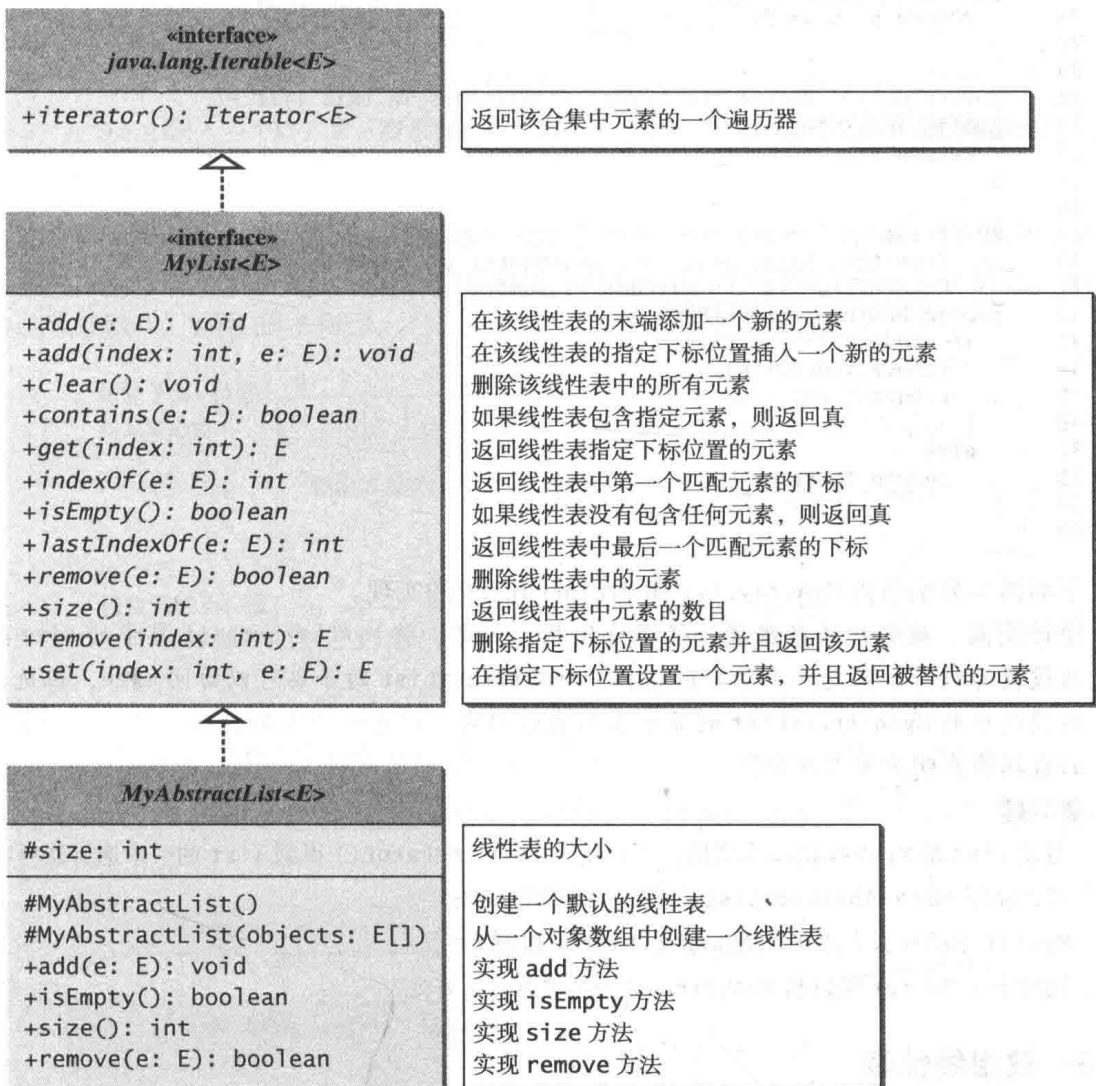


图 24-3 MyList 定义了操作线性表的许多方法。MyAbstractList 提供 MyList 接口的部分实现

程序清单 24-2 MyAbstractList.java

```

1  public abstract class MyAbstractList<E> implements MyList<E> {
2      protected int size = 0; // The size of the list
3
4      /** Create a default list */
5      protected MyAbstractList() {

```

```

6    }
7
8    /** Create a list from an array of objects */
9    protected MyAbstractList(E[] objects) {
10        for (int i = 0; i < objects.length; i++)
11            add(objects[i]);
12    }
13
14    @Override /** Add a new element at the end of this list */
15    public void add(E e) {
16        add(size, e);
17    }
18
19    @Override /** Return true if this list doesn't contain any elements */
20    public boolean isEmpty() {
21        return size == 0;
22    }
23
24    @Override /** Return the number of elements in this list */
25    public int size() {
26        return size;
27    }
28
29    @Override /** Remove the first occurrence of the element e
30     * from this list. Shift any subsequent elements to the left.
31     * Return true if the element is removed. */
32    public boolean remove(E e) {
33        if (indexOf(e) >= 0) {
34            remove(indexOf(e));
35            return true;
36        }
37        else
38            return false;
39    }
40 }

```

下面两节分别给出 `MyArrayList` 和 `MyLinkedList` 的实现。

{ } 设计指南：被保护的数据域一般很少使用，但是，将 `MyAbstractList` 类中的 `size` 数据域设置为被保护的，是一个很好的选择。`MyAbstractList` 的子类可以访问 `size`，但是，在不同包中的 `MyAbstractList` 的非子类不能访问它。作为一个常用规则，可以将抽象类中的数据域声明为被保护的。

✓ 复习题

- 24.1 假设 `list` 是 `MyList` 的一个实例，可以使用 `list.iterator()` 得到 `list` 的一个遍历器吗？
- 24.2 可以使用 `new MyAbstractList()` 创建一个线性表吗？
- 24.3 `MyList` 中的什么方法在 `MyAbstractList` 中被覆盖？
- 24.4 同时定义 `MyList` 接口和 `MyAbstractList` 类有什么好处？

24.3 数组线性表

🔑 要点提示：数组线性表采用数组来实现。

数组是一种大小固定的数据结构。数组一旦创建之后，它的大小就无法改变。尽管如此，仍然可以使用数组来实现动态的数据结构。处理的方法是，当数组不能再存储线性表中的新元素时，创建一个更大的新数组来替换当前数组。

初始化时，用默认大小创建一个类型为 `E[]` 的数组 `data`。向数组中插入一个新元素时，首先确认数组是否有足够的空间。若数组的空间不够，则创建大小为当前数组两倍的新数

组，然后将当前数组中的元素复制到新的数组中。现在，新数组就变成了当前数组。在指定下标处插入一个新元素之前，必须将指定下标后面的所有元素都向右移动一个位置并且将该线性表的大小增加 1，如图 24-4 所示。

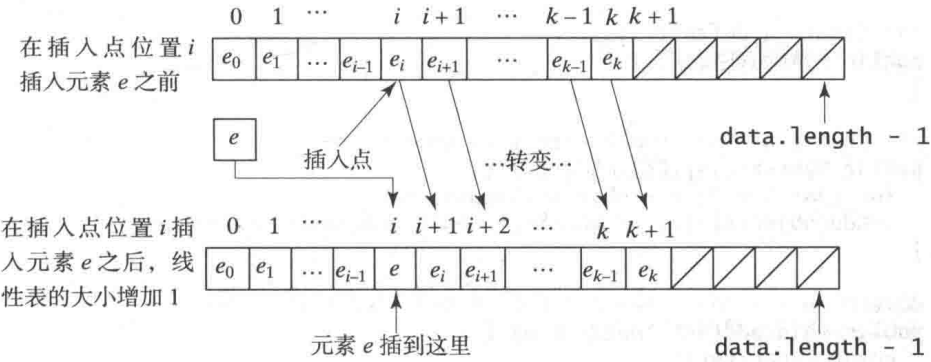


图 24-4 在数组中插入一个新元素，要求插入点之后的所有元素都向右移动一个位置，以便在插入点插入新元素

注意：该数据数组的类型是 `E[]` 类型，所以数组中每个元素实际存储的是对象的引用。

删除指定下标处的一个元素时，应该将该下标后面的元素都向左移动一个位置，并将线性表的大小减 1，如图 24-5 所示。

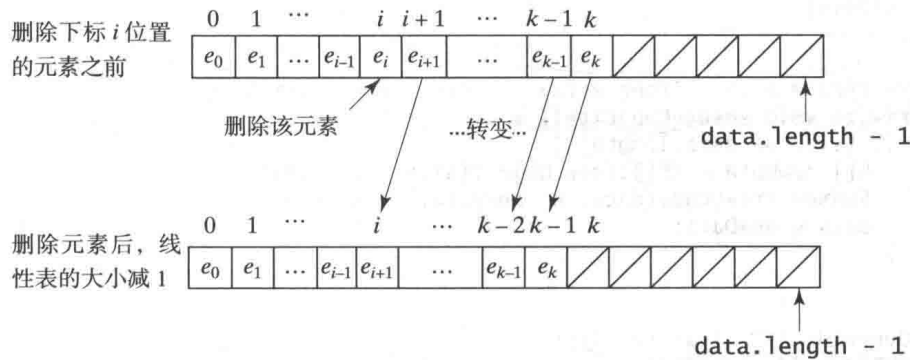


图 24-5 从数组中删除一个元素，要求删除点之后的元素都向左移动一个位置

`MyArrayList` 使用数组来实现 `MyAbstractList`，如图 24-6 所示。它的实现在程序清单 24-3 中给出。

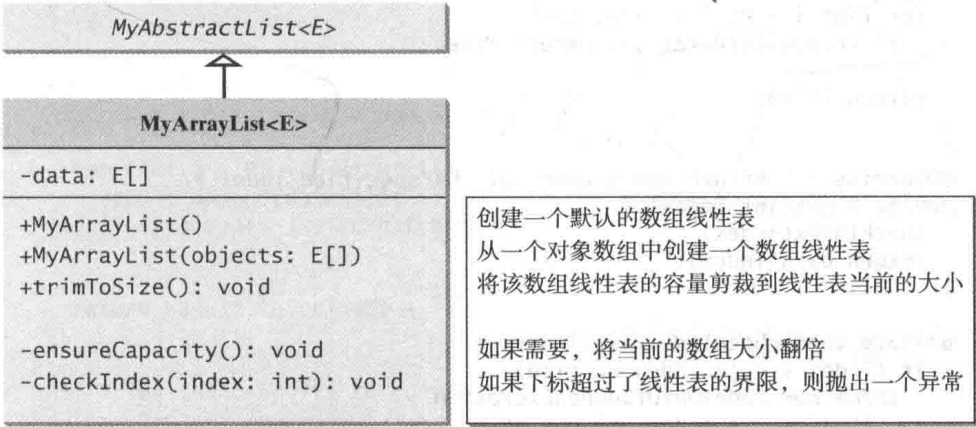


图 24-6 `MyArrayList` 使用数组实现线性表

程序清单 24-3 MyArrayList.java

```

1  public class MyArrayList<E> extends MyAbstractList<E> {
2      public static final int INITIAL_CAPACITY = 16;
3      private E[] data = (E[]) new Object[INITIAL_CAPACITY];
4
5      /** Create a default list */
6      public MyArrayList() {
7      }
8
9      /** Create a list from an array of objects */
10     public MyArrayList(E[] objects) {
11         for (int i = 0; i < objects.length; i++)
12             add(objects[i]); // Warning: don't use super(objects)!
13     }
14
15     @Override /** Add a new element at the specified index */
16     public void add(int index, E e) {
17         ensureCapacity();
18
19         // Move the elements to the right after the specified index
20         for (int i = size - 1; i >= index; i--)
21             data[i + 1] = data[i];
22
23         // Insert new element to data[index]
24         data[index] = e;
25
26         // Increase size by 1
27         size++;
28     }
29
30     /** Create a new larger array, double the current size + 1 */
31     private void ensureCapacity() {
32         if (size >= data.length) {
33             E[] newData = (E[]) new Object[size * 2 + 1];
34             System.arraycopy(data, 0, newData, 0, size);
35             data = newData;
36         }
37     }
38
39     @Override /** Clear the list */
40     public void clear() {
41         data = (E[]) new Object[INITIAL_CAPACITY];
42         size = 0;
43     }
44
45     @Override /** Return true if this list contains the element */
46     public boolean contains(E e) {
47         for (int i = 0; i < size; i++)
48             if (e.equals(data[i])) return true;
49
50         return false;
51     }
52
53     @Override /** Return the element at the specified index */
54     public E get(int index) {
55         checkIndex(index);
56         return data[index];
57     }
58
59     private void checkIndex(int index) {
60         if (index < 0 || index >= size)
61             throw new IndexOutOfBoundsException
62                 ("index " + index + " out of bounds");

```

```
63     }
64
65     @Override /** Return the index of the first matching element
66      * in this list. Return -1 if no match. */
67     public int indexOf(E e) {
68         for (int i = 0; i < size; i++)
69             if (e.equals(data[i])) return i;
70
71         return -1;
72     }
73
74     @Override /** Return the index of the last matching element
75      * in this list. Return -1 if no match. */
76     public int lastIndexOf(E e) {
77         for (int i = size - 1; i >= 0; i--)
78             if (e.equals(data[i])) return i;
79
80         return -1;
81     }
82
83     @Override /** Remove the element at the specified position
84      * in this list. Shift any subsequent elements to the left.
85      * Return the element that was removed from the list. */
86     public E remove(int index) {
87         checkIndex(index);
88
89         E e = data[index];
90
91         // Shift data to the left
92         for (int j = index; j < size - 1; j++)
93             data[j] = data[j + 1];
94
95         data[size - 1] = null; // This element is now null
96
97         // Decrement size
98         size--;
99
100        return e;
101    }
102
103    @Override /** Replace the element at the specified position
104     * in this list with the specified element. */
105    public E set(int index, E e) {
106        checkIndex(index);
107        E old = data[index];
108        data[index] = e;
109        return old;
110    }
111
112    @Override
113    public String toString() {
114        StringBuilder result = new StringBuilder("[");
115
116        for (int i = 0; i < size; i++) {
117            result.append(data[i]);
118            if (i < size - 1) result.append(", ");
119        }
120
121        return result.toString() + "]";
122    }
123
124    /** Trims the capacity to current size */
125    public void trimToSize() {
```

```

126     if (size != data.length) {
127         E[] newData = (E[])(new Object[size]);
128         System.arraycopy(data, 0, newData, 0, size);
129         data = newData;
130     } // If size == capacity, no need to trim
131 }
132
133 @Override /** Override iterator() defined in Iterable */
134 public java.util.Iterator<E> iterator() {
135     return new ArrayListIterator();
136 }
137
138 private class ArrayListIterator
139     implements java.util.Iterator<E> {
140     private int current = 0; // Current index
141
142     @Override
143     public boolean hasNext() {
144         return (current < size);
145     }
146
147     @Override
148     public E next() {
149         return data[current++];
150     }
151
152     @Override
153     public void remove() {
154         MyArrayList.this.remove(current);
155     }
156 }
157 }

```

常量 `INITIAL_CAPACITY` (第 2 行) 用于创建一个初始数组 `data` (第 3 行)。由于泛型消除, 所以不能使用语法 `new e[INITIAL_CAPACITY]` 创建泛型数组。为了规避这个限制, 第 3 行创建了一个 `Object` 类型的数组, 并将它转换为 `E[]` 类型。

注意, `MyArrayList` 中的第二个构造方法的实现和 `MyAbstractList` 的构造方法一样。可以用 `super(objects)` 替换第 11 ~ 12 行吗? 参见复习题 24.8 来寻找答案。

`add(int index, E e)` 方法 (第 16 ~ 28 行) 将元素 `e` 添加到数组的指定下标 `index` 处。该方法首先调用 `ensureCapacity()` 方法 (第 17 行), 以确保数组中还有存储新元素的空间。在插入新元素之前, 将指定下标后面的所有元素都向右移动一个位置 (第 20 ~ 21 行)。在数组中添加新元素之后, 数组的大小 `size` 也随之加 1 (第 27 行)。注意, `MyAbstractList` 中的变量 `size` 被定义为 `protected`, 所以它可以被 `MyArrayList` 访问。

`ensureCapacity()` 方法 (第 31 ~ 37 行) 用来检验数组是否已满。如果数组已满, 则创建一个容量为当前数组大小两倍 + 1 的新数组, 并使用 `System.arraycopy` 方法将当前数组的所有元素复制到新数组中, 再把新数组设为当前数组。

`clear()` 方法 (第 40 ~ 43 行) 创建一个具有初始大小为 `INITIAL_CAPACITY` 的全新数组, 并设置变量 `size` 为 0。如果删除第 41 行, 类可以运行, 但是将会产生内存泄露, 因为即使已经不再被需要, 但是元素依然在数组中。通过创建一个新数组并且将其赋值给 `data`, 老的数组和保存在老数组中的元素变成了垃圾, 将自动被 JVM 所收集。

`contains(E e)` 方法 (第 46 ~ 51 行) 使用 `equals` 方法将元素 `e` 与数组中的所有元素逐一比较, 以判断数组中是否包含元素 `e`。

`get(int index)` 方法 (第 54 ~ 57 行) 检查 `index` 是否在范围内, 并且如果 `index` 在范围内, 则返回 `data[index]`。

`checkIndex(int index)` 方法 (第 59 ~ 63 行) 检查 `index` 是否在范围内, 如果不在, 方法抛出一个 `IndexOutOfBoundsException` (第 61 行)。

`indexOf(E e)` 方法 (第 67 ~ 72 行) 从第一个元素开始, 将元素 `e` 与数组中的每一个元素逐一比较。如果匹配, 则返回匹配元素的下标; 否则, 返回 `-1`。

`lastIndexOf(E e)` 方法 (第 76 ~ 81 行) 从最后一个元素开始, 将元素 `e` 与数组的每一个元素逐一比较。如果匹配, 则返回匹配元素的下标; 否则, 返回 `-1`。

`remove(int index)` 方法 (第 86 ~ 101 行) 将指定下标之后所有元素向左移动一个位置 (第 92 ~ 93 行), 并将数组大小 `size` 减 1 (第 98 行)。最后一个元素不再使用, 设置为 `null` (第 95 行)。

`set(int index, E e)` 方法 (第 105 ~ 110 行) 只是简单地将 `e` 赋给 `data[index]`, 将数组中指定下标处的元素用 `e` 替换。

`toString()` 方法 (第 113 ~ 122 行) 覆盖 `Object` 类中的 `toString` 方法, 返回一个表示线性表中所有元素的字符串。

`trimToSize()` 方法 (第 125 ~ 131 行) 创建一个新数组, 它的大小与当前数组线性表的大小匹配 (第 127 行), 使用 `System.arraycopy` 方法将当前数组复制到新的数组中 (第 128 行), 然后将新数组设置为当前数组 (第 129 行)。注意, 如果 `size == capacity`, 就无须裁剪数组的大小。

`java.lang.Iterable` 接口中定义的 `iterator()` 方法被实现为返回一个 `java.util.Iterator` 的实例 (第 134 ~ 136 行)。`ArrayListIterator` 类实现了 `Iterator` 中的方法 `hasNext`、`next` 以及 `remove` (第 143 ~ 155 行)。它使用 `current` 来标识被遍历的元素的当前位置 (第 140 行)。

程序清单 24-4 给出一个使用 `MyArrayList` 创建线性表的例子。它使用 `add` 方法来给线性表添加一个字符串, 并使用 `remove` 方法来删除字符串。由于 `MyArrayList` 实现了 `Iterable`, 元素可以使用一个 `foreach` 循环来进行遍历 (第 35 ~ 36 行)。

程序清单 24-4 Test My Array List.java

```
1 public class TestMyArrayList {
2     public static void main(String[] args) {
3         // Create a list
4         MyList<String> list = new MyArrayList<String>();
5
6         // Add elements to the list
7         list.add("America"); // Add it to the list
8         System.out.println("(1) " + list);
9
10        list.add(0, "Canada"); // Add it to the beginning of the list
11        System.out.println("(2) " + list);
12
13        list.add("Russia"); // Add it to the end of the list
14        System.out.println("(3) " + list);
15
16        list.add("France"); // Add it to the end of the list
17        System.out.println("(4) " + list);
18
19        list.add(2, "Germany"); // Add it to the list at index 2
20        System.out.println("(5) " + list);
21    }
22 }
```

```

21
22     list.add(5, "Norway"); // Add it to the list at index 5
23     System.out.println("(6) " + list);
24
25     // Remove elements from the list
26     list.remove("Canada"); // Same as list.remove(0) in this case
27     System.out.println("(7) " + list);
28
29     list.remove(2); // Remove the element at index 2
30     System.out.println("(8) " + list);
31
32     list.remove(list.size() - 1); // Remove the last element
33     System.out.print("(9) " + list + "\n(10) ");
34
35     for (String s: list)
36         System.out.print(s.toUpperCase() + " ");
37     }
38 }

```

```

(1) [America]
(2) [Canada, America]
(3) [Canada, America, Russia]
(4) [Canada, America, Russia, France]
(5) [Canada, America, Germany, Russia, France]
(6) [Canada, America, Germany, Russia, France, Norway]
(7) [America, Germany, Russia, France, Norway]
(8) [America, Germany, France, Norway]
(9) [America, Germany, France]
(10) AMERICA GERMANY FRANCE

```

✓ 复习题

- 24.5 数组数据类型的局限性是什么？
- 24.6 `MyArrayList` 是使用数组来实现的，而数组是一种大小固定的数据结构。那么为什么说 `MyArrayList` 是动态的数据结构呢？
- 24.7 下面的语句执行后，给出 `MyArrayList` 中数组的长度。

```

1  MyArrayList<Double> list = new MyArrayList<>();
2  list.add(1.5);
3  list.trimToSize();
4  list.add(3.4);
5  list.add(7.4);
6  list.add(17.4);

```

- 24.8 如果程序清单 24-3 中的第 11 ~ 12 行

```

for (int i = 0; i < objects.length; i++)
    add(objects[i]);

```

被下面的语句代替

```
super(objects);
```

或者被下面的语句代替

```

data = objects;
size = objects.length;

```

会出现什么错误？

- 24.9 如果将程序清单 24-3 中第 33 行的代码从

```
E[] newData = (E[])(new Object[size * 2 + 1]);
```

改为

```
E[] newData = (E[])(new Object[size * 2]);
```

程序就是错误的。你能找出原因吗?

24.10 如果 41 行的以下代码被删除, `MyArrayList` 类会有内存泄露么?

```
data = (E[])new Object[INITIAL_CAPACITY];
```

24.11 如果下标越界, `get(index)` 方法调用 `checkIndex(index)` 方法 (程序清单 24-3 第 59~63 行) 会抛出 `IndexOutOfBoundsException`。假设 `add(index, e)` 如下实现:

```
public void add(int index, E e) {
    checkIndex(index);

    // Same as lines 17-27 in Listing 24.3 MyArrayList.java
}
```

那么运行下面的代码会发生什么情况?

```
MyArrayList<String> list = new MyArrayList<>();
list.add("New York");
```

24.4 链表

🔑 要点提示: 链表采用链接结构实现。

由于 `MyArrayList` 是用数组实现的, 所以 `get(int index)` 和 `set(int index, E e)` 方法可以通过下标访问和修改元素, 也可以用 `add(E e)` 方法在线性表末尾添加元素, 它们是高效的。但是, `add(int index, E e)` 和 `remove(int index)` 方法的效率很低, 因为这两个方法需要移动潜在的大量元素。为提高在表中开始位置添加和删除元素的效率, 可以采用链式结构来实现线性表。

24.4.1 结点

链表中的每个元素都包含一个称为结点 (node) 的结构。当向链表中加入一个新的元素时, 就会产生一个包含它的结点。每个结点都和它的相邻结点相链接, 如图 24-7 所示。

结点可以按如下方式定义为一个类:

```
class Node<E> {
    E element;
    Node<E> next;

    public Node(E e) {
        element = e;
    }
}
```

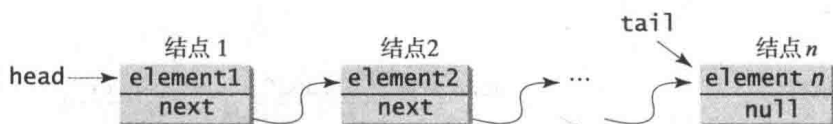


图 24-7 链表由链接在一起的任意多个结点构成

变量 `head` 指向线性表的第一个结点, 而变量 `tail` 指向最后一个结点。如果线性表为空, `head` 和 `tail` 这两个变量均为 `null`。下面就是一个创建存储三个结点的链表的例子, 其

中每个结点存储一个字符串元素。

步骤 1: 声明 `head` 和 `tail`。

```
Node<String> head = null;      线性表现在为空
Node<String> tail = null;
```

`head` 和 `tail` 都为 `null`。该线性表为空。

步骤 2: 创建第一个结点并将它追加到线性表中, 如图 24-8 所示。在将第一个结点插入线性表之后, `head` 和 `tail` 都指向这个结点。

```
head = new Node<>("Chicago");  第一个结点插入后
tail = head;
```

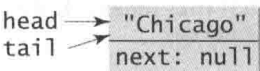
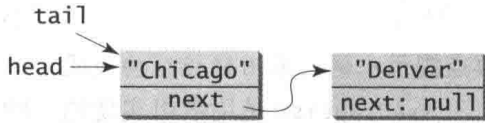


图 24-8 向线性表追加第一个结点。头和尾结点都指向该结点

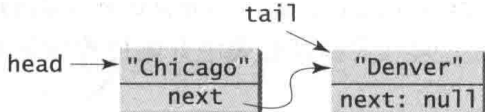
步骤 3: 创建第二个结点并将它追加到线性表中, 如图 24-9a 所示。为了将第二个结点追加到线性表中, 需要将新结点和第一个结点链接起来, 现在, 新结点就是尾结点。所以, 应该移动 `tail`, 使它指向该新结点, 如图 24-9b 所示。

```
tail.next = new Node<>("Denver");
```



a)

```
tail = tail.next;
```

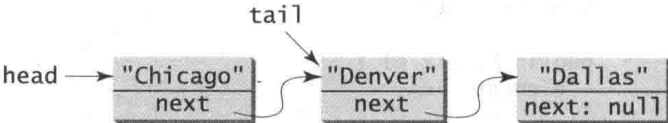


b)

图 24-9 向线性表追加第二个结点。尾结点现在指向这个新的结点

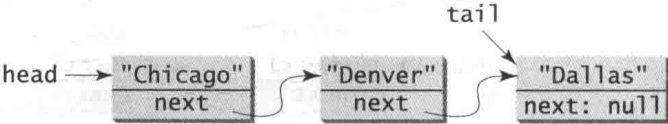
步骤 4: 创建第三个结点并将它追加到线性表中, 如图 24-10a 所示。为了向线性表追加新的结点, 链接新结点和线性表当前的最后一个结点。现在, 新结点就是尾结点。所以, 应该移动 `tail`, 使它指向该新结点, 如图 24-10b 所示。

```
tail.next = new Node<>("Dallas");
```



a)

```
tail = tail.next;
```



b)

图 24-10 向线性表追加第三个结点

每个结点都包含元素和一个名为 `next` 的数据域, `next` 指向下一个元素。如果结点是线

性表中的最后一个，那么它的指针数据域 `next` 所包含的值是 `null`。可以使用这个特性来检测某结点是否是最后的结点。例如，可以编写下面的循环遍历线性表中的所有结点。

```

1 Node current = head;
2 while (current != null) {
3     System.out.println(current.element);
4     current = current.next;
5 }

```

初始状态时，变量 `current` 指向线性表的第一个结点（第 1 行）。在循环中，获取当前结点的元素（第 3 行），然后 `current` 指向下一个结点（第 4 行）。循环持续到当前结点为 `null` 时为止。

24.4.2 MyLinkedList 类

`MyLinkedList` 类使用链式结构实现动态线性表，它继承自 `MyAbstractList` 类。此外，它还提供 `addFirst`、`addLast`、`removeFirst`、`removeLast`、`getFirst` 和 `getLast` 方法，如图 24-11 所示。

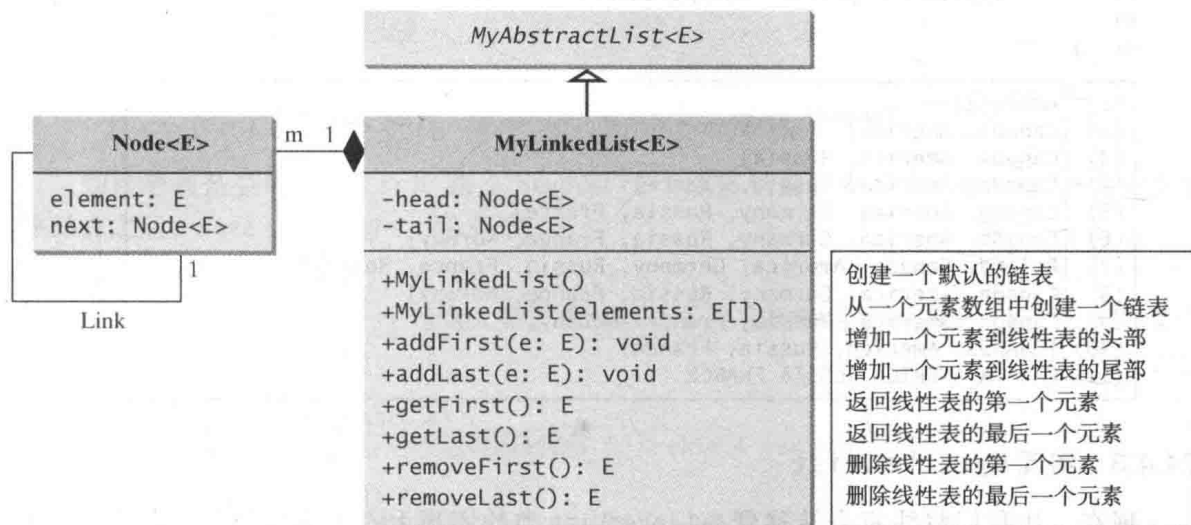


图 24-11 `MyLinkedList` 使用链接在一起的结点实现链表

假设已经实现了这个类，程序清单 24-5 给出使用该类的测试程序。

程序清单 24-5 TestMyLinkedList.java

```

1 public class TestMyLinkedList {
2     /** Main method */
3     public static void main(String[] args) {
4         // Create a list for strings
5         MyLinkedList<String> list = new MyLinkedList<>();
6
7         // Add elements to the list
8         list.add("America"); // Add it to the list
9         System.out.println("(1) " + list);
10
11        list.add(0, "Canada"); // Add it to the beginning of the list
12        System.out.println("(2) " + list);
13
14        list.add("Russia"); // Add it to the end of the list
15        System.out.println("(3) " + list);
16
17        list.addLast("France"); // Add it to the end of the list
18        System.out.println("(4) " + list);

```

```

19
20 list.add(2, "Germany"); // Add it to the list at index 2
21 System.out.println("(5) " + list);
22
23 list.add(5, "Norway"); // Add it to the list at index 5
24 System.out.println("(6) " + list);
25
26 list.add(0, "Poland"); // Same as list.addFirst("Poland")
27 System.out.println("(7) " + list);
28
29 // Remove elements from the list
30 list.remove(0); // Same as list.remove("Poland") in this case
31 System.out.println("(8) " + list);
32
33 list.remove(2); // Remove the element at index 2
34 System.out.println("(9) " + list);
35
36 list.remove(list.size() - 1); // Remove the last element
37 System.out.print("(10) " + list + "\n(11) ");
38
39 for (String s: list)
40     System.out.print(s.toUpperCase() + " ");
41 }
42 }

```

```

(1) [America]
(2) [Canada, America]
(3) [Canada, America, Russia]
(4) [Canada, America, Russia, France]
(5) [Canada, America, Germany, Russia, France]
(6) [Canada, America, Germany, Russia, France, Norway]
(7) [Poland, Canada, America, Germany, Russia, France, Norway]
(8) [Canada, America, Germany, Russia, France, Norway]
(9) [Canada, America, Russia, France, Norway]
(10) [Canada, America, Russia, France]
(11) CANADA AMERICA RUSSIA FRANCE

```

24.4.3 实现 MyLinkedList

现在，让我们将注意力转移到 MyLinkedList 类的实现上。下面将讨论如何实现方法 addFirst、addLast、add(index,e)、removeFirst、removeLast 和 remove(index)，并且将 MyLinkedList 类中的其他方法留作练习题。

实现 addFirst(e) 方法

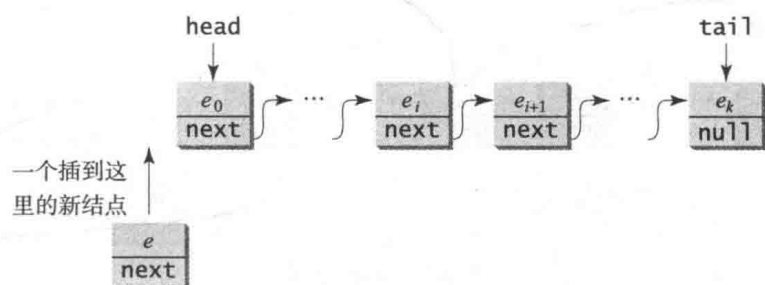
addFirst(e) 方法创建一个包含元素 e 的新结点。该新结点就成为链表的第一个结点。该方法可以如下实现：

```

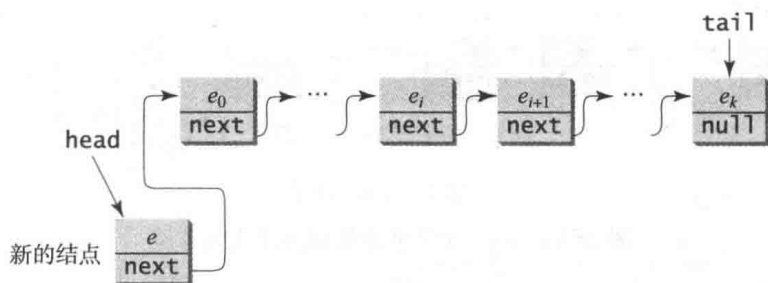
1 public void addFirst(E e) {
2     Node<E> newNode = new Node<>(e); // Create a new node
3     newNode.next = head; // link the new node with the head
4     head = newNode; // head points to the new node
5     size++; // Increase list size
6
7     if (tail == null) // The new node is the only node in list
8         tail = head;
9 }

```

addFirst(e) 方法创建一个新结点来存储元素 (第 2 行)，并将该结点插入链表的起始位置 (第 3 行)，如图 24-12a 所示。在插入之后，head 应该指向该新元素结点 (第 4 行)，如图 24-12b 所示。



a) 插入一个新结点前



b) 插入一个新结点后

图 24-12 将一个新元素添加到链表的起始位置

如果链表是空的（第 7 行），那么 head 和 tail 都将指向该新结点（第 8 行）。在创建完该结点之后，size 应该增加 1（第 5 行）。

实现 addLast(e) 方法

addLast(e) 方法创建一个包含元素的新结点，并将它插到链表的末尾。该方法可以如下实现：

```

1 public void addLast(E e) {
2     Node<E> newNode = new Node<>(e); // Create a new node for e
3
4     if (tail == null) {
5         head = tail = newNode; // The only node in list
6     }
7     else {
8         tail.next = newNode; // Link the new node with the last node
9         tail = tail.next; // tail now points to the last node
10    }
11
12    size++; // Increase size
13 }

```

addLast(e) 方法创建一个新结点来存储元素（第 2 行），并且将它追加到链表的末尾。考虑以下两种情况：

- 1) 如果链表为空（第 4 行），那么 head 和 tail 都将指向该新结点（第 5 行）。
- 2) 否则，将该结点和该链表的最后一个结点相链接（第 8 行）。现在，tail 应该指向该新结点（第 9 行）。图 24-13a 和图 23-14b 演示了插入前后的包含元素 e 的新结点。

不论哪种情况，在创建一个结点后，链表的大小都加 1（第 12 行）。

实现 add(index, e) 方法

add(index, e) 方法将一个元素插到链表的指定下标处。该方法可以如下实现：

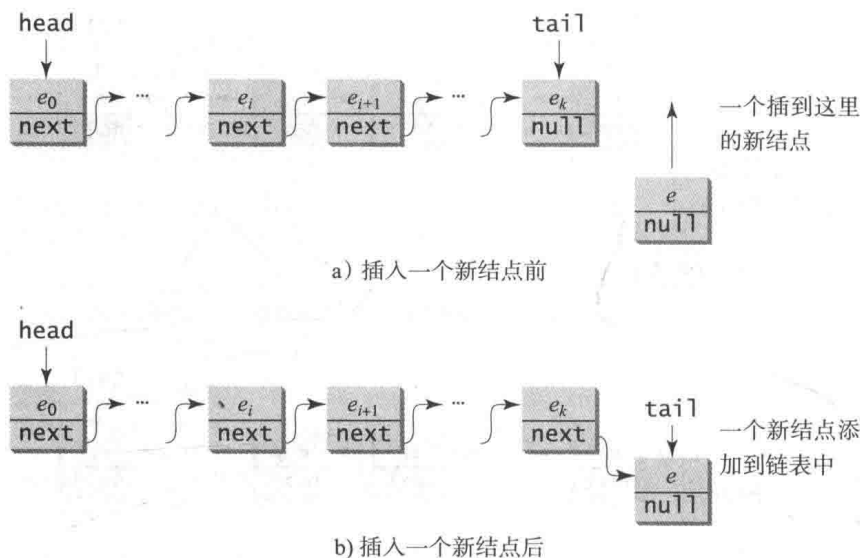


图 24-13 将一个新元素添加到链表的末尾

```

1 public void add(int index, E e) {
2     if (index == 0) addFirst(e); // Insert first
3     else if (index >= size) addLast(e); // Insert last
4     else { // Insert in the middle
5         Node<E> current = head;
6         for (int i = 1; i < index; i++)
7             current = current.next;
8         Node<E> temp = current.next;
9         current.next = new Node<E>(e);
10        (current.next).next = temp;
11        size++;
12    }
13 }

```

将一个元素插入链表中时，会出现以下三种情况：

1) 当指定下标 `index` 为 0 时，调用 `addFirst(e)` 方法（第 2 行）将该元素插到链表的起始位置；

2) 当 `index` 大于或等于链表的大小 `size` 时，调用 `addLast(e)` 方法（第 3 行）将元素 `e` 添加到链表的末尾；

3) 否则，创建一个新结点来存储新元素，然后定位它的插入位置。新的结点应该插入到结点 `current` 和 `temp` 之间，如图 24-14a 所示。该方法将新结点赋给 `current.next`，并将 `temp` 赋给新结点的 `next`，如图 24-14b 所示。现在，链表的大小也要增加 1（第 11 行）。

实现 `removeFirst()` 方法

`removeFirst()` 方法从链表中删除第一个元素。该方法可以如下实现：

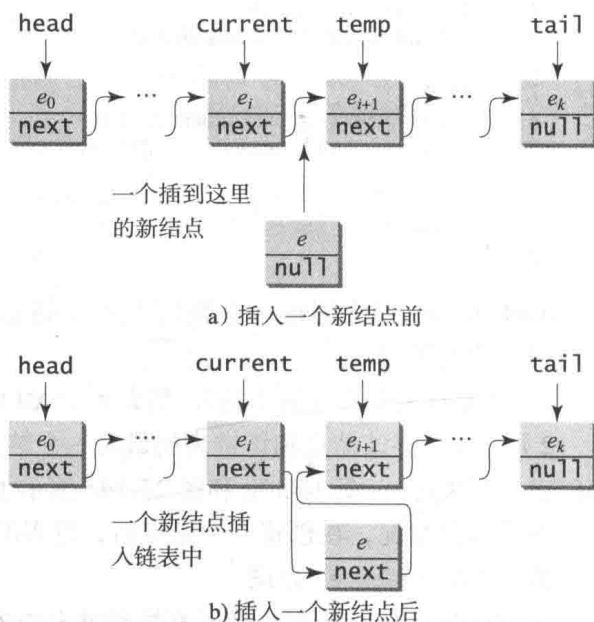


图 24-14 将新元素插到链表的中间

```

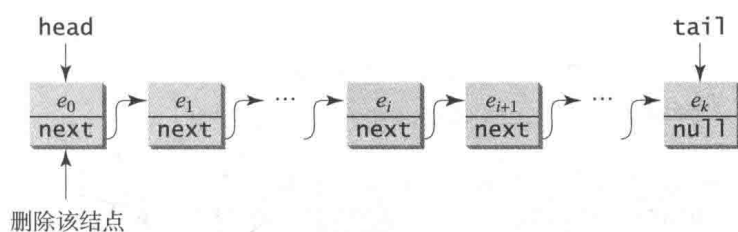
1 public E removeFirst() {
2     if (size == 0) return null; // Nothing to delete
3     else {
4         Node<E> temp = head; // Keep the first node temporarily
5         head = head.next; // Move head to point to next node
6         size--; // Reduce size by 1
7         if (head == null) tail = null; // List becomes empty
8         return temp.element; // Return the deleted element
9     }
10 }

```

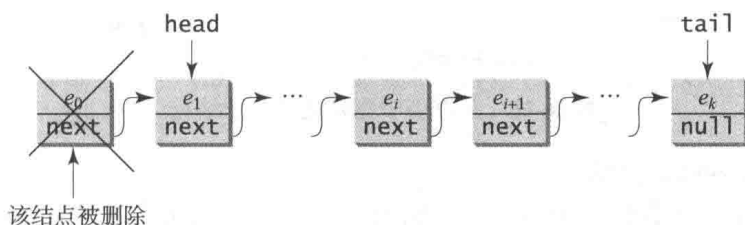
考虑以下两种情况：

1) 如果该链表为空，它就没有什么可删除的，因此返回 null (第 2 行)。

2) 否则，通过将 head 指向第二个结点以从链表中删除第一个结点。图 24-15a 和图 24-15b 展示了删除之前和删除之后的链表。在删除之后，链表的大小减 1 (第 6 行)。如果链表为空，那么在删除该元素之后，tail 应该设置为 null (第 7 行)。



a) 删除一个结点前



b) 删除一个结点后

图 24-15 从链表中删除第一个结点

实现 removeLast() 方法

`removeLast()` 方法从链表中删除最后一个元素。该方法可以如下实现：

```

1 public E removeLast() {
2     if (size == 0) return null; // Nothing to remove
3     else if (size == 1) { // Only one element in the list
4         Node<E> temp = head;
5         head = tail = null; // list becomes empty
6         size = 0;
7         return temp.element;
8     }
9     else {
10        Node<E> current = head;
11
12        for (int i = 0; i < size - 2; i++)
13            current = current.next;
14
15        Node<E> temp = tail;
16        tail = current;
17        tail.next = null;
18        size--;
19        return temp.element;
20    }
21 }

```

考虑以下 3 种情况：

1) 如果链表为空，则返回 `null` (第 2 行)。

2) 如果链表只有一个结点，该结点就被销毁，`head` 和 `tail` 都会成为 `null` (第 5 行)。删除后大小变为 0 (第 6 行)，删除结点中的元素值被返回 (第 7 行)。

3) 否则，最后一个结点被销毁 (第 17 行)，`tail` 重新定位到指向倒数第二个结点，如图 24-16a 和图 24-16b 显示了删除前后的最后一个结点。在删除之后，链表的大小减 1 (第 18 行)，然后返回被删除结点的元素值 (第 19 行)。

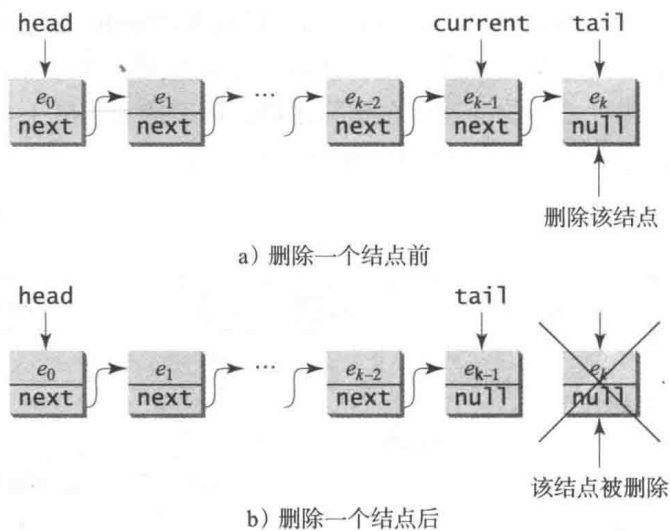


图 24-16 从链表中删除最后一个结点

实现 `remove(index)` 方法

`remove(index)` 方法找到指定下标处的结点，然后将它删除。该方法可以如下实现：

```

1 public E remove(int index) {
2     if (index < 0 || index >= size) return null; // Out of range
3     else if (index == 0) return removeFirst(); // Remove first
4     else if (index == size - 1) return removeLast(); // Remove last
5     else {
6         Node<E> previous = head;
7
8         for (int i = 1; i < index; i++) {
9             previous = previous.next;
10        }
11
12        Node<E> current = previous.next;
13        previous.next = current.next;
14        size--;
15        return current.element;
16    }
17 }

```

考虑以下 4 种情况：

1) 如果 `index` 超出链表的范围 (即 `index < 0 || index >= size`)，则返回 `null` (第 2 行)。

2) 如果 `index` 为 0，则调用 `removeFirst()` 方法删除链表的第一个结点 (第 3 行)。

3) 当 `index` 为 `size-1` 时，则调用 `removeLast()` 方法删除链表的最后一个结点 (第 4 行)。

4) 否则, 找到指定 index 位置的结点, 用 current 指向这个结点, 用 previous 指向该结点的前一个结点, 如图 24-17a 所示。将 current.next 赋给 previous.next 以删除当前结点, 如图 24-17b 所示。

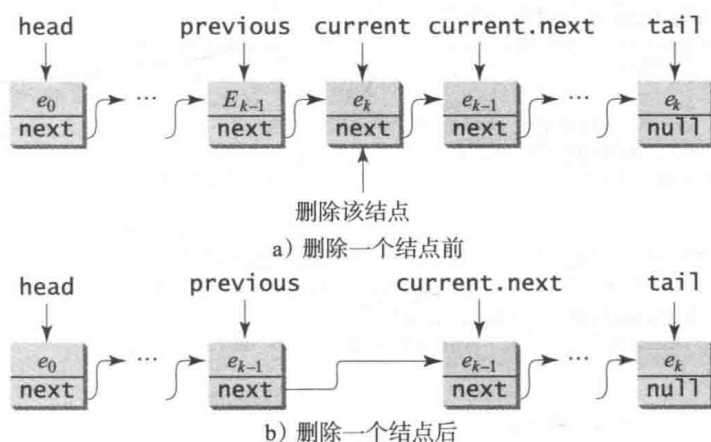


图 24-17 从链表中删除一个结点

程序清单 24-6 给出了 MyLinkedList 的实现。这里, 忽略方法 get(index)、indexOf(e)、lastIndexOf(e)、contains(e) 和 set(index,e) 的实现, 将它们留作练习题。实现了 java.lang.Iterable 接口中定义的方法 iterator(), 返回一个 java.util.Iterator 的实例 (第 126 ~ 128 行)。LinkedListIterator 类实现了 Iterator 接口, 具有 hasNext、next, 以及 remove 等具体方法 (第 134 ~ 149 行)。该实现使用 current 来指向被遍历的元素的当前位置 (第 132 行)。最开始, current 指向线性表的头部。

程序清单 24-6 MyLinkedList.java

```

1 public class MyLinkedList<E> extends MyAbstractList<E> {
2     private Node<E> head, tail;
3
4     /** Create a default list */
5     public MyLinkedList() {
6     }
7
8     /** Create a list from an array of objects */
9     public MyLinkedList(E[] objects) {
10         super(objects);
11     }
12
13     /** Return the head element in the list */
14     public E getFirst() {
15         if (size == 0) {
16             return null;
17         }
18         else {
19             return head.element;
20         }
21     }
22
23     /** Return the last element in the list */
24     public E getLast() {
25         if (size == 0) {
26             return null;
27         }
28         else {

```



```

29     return tail.element;
30 }
31 }
32
33 /** Add an element to the beginning of the list */
34 public void addFirst(E e) {
35     // Implemented in Section 24.4.3.1, so omitted here
36 }
37
38 /** Add an element to the end of the list */
39 public void addLast(E e) {
40     // Implemented in Section 24.4.3.2, so omitted here
41 }
42
43 @Override /** Add a new element at the specified index
44  * in this list. The index of the head element is 0 */
45 public void add(int index, E e) {
46     // Implemented in Section 24.4.3.3, so omitted here
47 }
48
49 /** Remove the head node and
50  * return the object that is contained in the removed node. */
51 public E removeFirst() {
52     // Implemented in Section 24.4.3.4, so omitted here
53 }
54
55 /** Remove the last node and
56  * return the object that is contained in the removed node. */
57 public E removeLast() {
58     // Implemented in Section 24.4.3.5, so omitted here
59 }
60
61 @Override /** Remove the element at the specified position in this
62  * list. Return the element that was removed from the list. */
63 public E remove(int index) {
64     // Implemented earlier in Section 24.4.3.6, so omitted here
65 }
66
67 @Override
68 public String toString() {
69     StringBuilder result = new StringBuilder("");
70
71     Node<E> current = head;
72     for (int i = 0; i < size; i++) {
73         result.append(current.element);
74         current = current.next;
75         if (current != null) {
76             result.append(", "); // Separate two elements with a comma
77         }
78         else {
79             result.append("]"); // Insert the closing ] in the string
80         }
81     }
82
83     return result.toString();
84 }
85
86 @Override /** Clear the list */
87 public void clear() {
88     size = 0;
89     head = tail = null;
90 }
91
92 @Override /** Return true if this list contains the element e */

```

```

93  public boolean contains(E e) {
94      System.out.println("Implementation left as an exercise");
95      return true;
96  }
97
98  @Override /** Return the element at the specified index */
99  public E get(int index) {
100      System.out.println("Implementation left as an exercise");
101      return null;
102  }
103
104  @Override /** Return the index of the head matching element
105   * in this list. Return -1 if no match. */
106  public int indexOf(E e) {
107      System.out.println("Implementation left as an exercise");
108      return 0;
109  }
110
111  @Override /** Return the index of the last matching element
112   * in this list. Return -1 if no match. */
113  public int lastIndexOf(E e) {
114      System.out.println("Implementation left as an exercise");
115      return 0;
116  }
117
118  @Override /** Replace the element at the specified position
119   * in this list with the specified element. */
120  public E set(int index, E e) {
121      System.out.println("Implementation left as an exercise");
122      return null;
123  }
124
125  @Override /** Override iterator() defined in Iterable */
126  public java.util.Iterator<E> iterator() {
127      return new LinkedListIterator();
128  }
129
130  private class LinkedListIterator
131      implements java.util.Iterator<E> {
132      private Node<E> current = head; // Current index
133
134      @Override
135      public boolean hasNext() {
136          return (current != null);
137      }
138
139      @Override
140      public E next() {
141          E e = current.element;
142          current = current.next;
143          return e;
144      }
145
146      @Override
147      public void remove() {
148          System.out.println("Implementation left as an exercise");
149      }
150  }
151
152  // This class is only used in LinkedList, so it is private.
153  // This class does not need to access any
154  // instance members of LinkedList, so it is defined static.
155  private static class Node<E> {
156      E element;

```

```
157     Node<E> next;
158
159     public Node(E element) {
160         this.element = element;
161     }
162 }
163 }
```

24.4.4 MyArrayList 和 MyLinkedList

可以使用 MyArrayList 和 MyLinkedList 来存储线性表。使用数组实现 MyArrayList，使用链表实现 MyLinkedList。MyArrayList 的开销比 MyLinkedList 小。但是，如果需要在线性表的开始位置插入和删除元素，那么 MyLinkedList 的效率会高一些。表 24-1 总结了 MyArrayList 和 MyLinkedList 中方法的复杂度。注意，MyArrayList 和 java.util.ArrayList 一样，而 MyLinkedList 和 java.util.LinkedList 一样。

表 24-1 MyArrayList 和 MyLinkedList 中方法的时间复杂度

方法	MyArrayList/ArrayList	MyLinkedList/LinkedList
add(e: E)	$O(1)$	$O(1)$
add(index: int, e: E)	$O(n)$	$O(n)$
clear()	$O(1)$	$O(1)$
contains(e: E)	$O(n)$	$O(n)$
get(index: int)	$O(1)$	$O(n)$
indexOf(e: E)	$O(n)$	$O(n)$
isEmpty()	$O(1)$	$O(1)$
lastIndexOf(e: E)	$O(n)$	$O(n)$
remove(e: E)	$O(n)$	$O(n)$
size()	$O(1)$	$O(1)$
remove(index: int)	$O(n)$	$O(n)$
set(index: int, e: E)	$O(n)$	$O(n)$
addFirst(e: E)	$O(n)$	$O(1)$
removeFirst()	$O(n)$	$O(1)$

24.4.5 链表的变体

前一节介绍的链表称为单链表 (singly linked list)。它包含一个指向线性表第一个结点的指针。每个结点都包含一个指针，该指针指向紧随其后的结点。在某些应用中，链表的几种变体是很有用的。

循环单链表 (circular, singly linked list) 除了链表中的最后一个结点的指针指回到第一个结点以外，其他都很像单链表，如图 24-18a 所示。注意，在循环单链表中不需要 tail。head 指向链表中的当前结点。插入和删除操作都在当前结点处。循环单链表的一个很好的应用是在以分时方式服务多个用户的操作系统中，系统会从循环链表中选择一个用户，确保分给他一小部分 CPU 时间，然后继续移动到链表中的下一个用户。

双向链表 (doubly linked list) 包含带两个指针的结点，一个指针指向下一个结点，而另一个指针指向前一个结点，如图 24-18b 所示。为方便起见，这两个指针分别称为前向指针 (forward pointer) 和后向指针 (backward pointer)。因此，双向链表既可以向前遍历，也可

以往后遍历。java.util.LinkedList 类使用双向链表实现，支持使用 ListIterator 向前或者往后遍历链表。

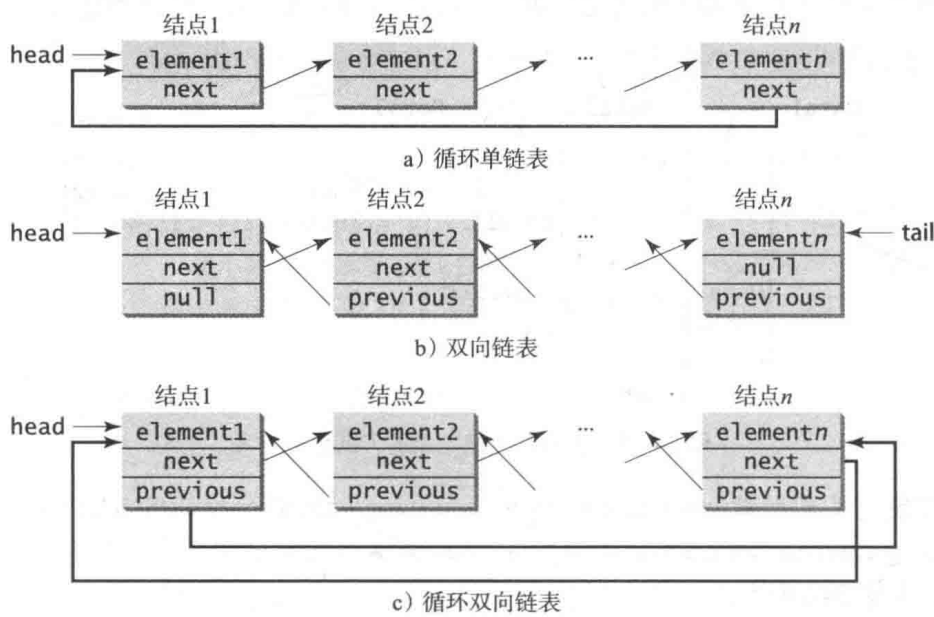


图 24-18 链表可表现为不同形式

循环双向链表 (circular, doubly linked list) 除了链表中最后一个结点的前向指针指向第一个结点，且第一个结点的后向指针指向最后一个结点以外，其他都和双向链表一样，如图 24-18c 所示。

这些链表的实现都留作练习题。

复习题

- 24.12 MyArrayList 和 MyLinkedList 都用于存储一个对象线性表。为什么我们两种线性表都需要？
- 24.13 绘图展示以下语句执行后的链表。

```
MyLinkedList<Double> list = new MyLinkedList<>();
list.add(1.5);
list.add(6.2);
list.add(3.4);
list.add(7.4);
list.remove(1.5);
list.remove(2);
```
- 24.14 MyLinkedList 中的 addFirst(e) 和 removeFirst() 的时间复杂度是多少？
- 24.15 假设你需要存储一个元素线性表。如果程序中的元素个数是固定的，应该使用什么数据结构？如果程序中的元素个数是变化的，应该使用什么数据结构？
- 24.16 如果需要在线性表的开始位置添加或删除元素，应该选择 MyArrayList 还是 MyLinkedList？如果线性表上面的大量操作都设计在一个给定位置来获取元素，应该选择 MyArrayList 还是 MyLinkedList？
- 24.17 使用条件表达式简化程序清单 24-6 中第 75 ~ 80 行的代码。

24.5 栈和队列

要点提示：可以使用数组线性表实现栈，使用链表实现队列。

栈可以看做是一种特殊类型的线性表，访问、插入和删除其中的元素只能在栈尾（栈顶）进行，如图 10-11 所示。队列表示一个等待的线性表，它也可以看做是一种特殊类型的线性表，元素只能从队列的末端（队列尾）插入，从开始端（队列头）访问和删除，如图 24-19 所示。

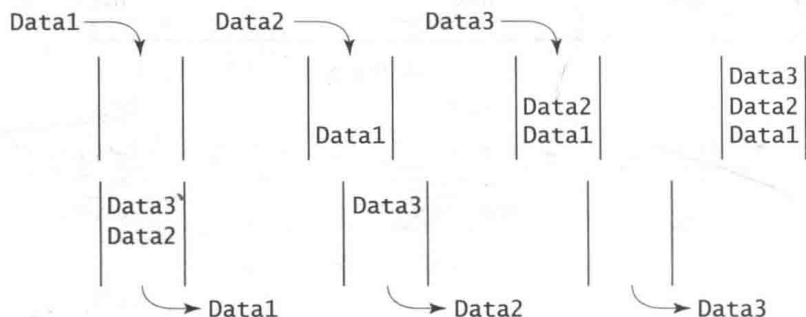
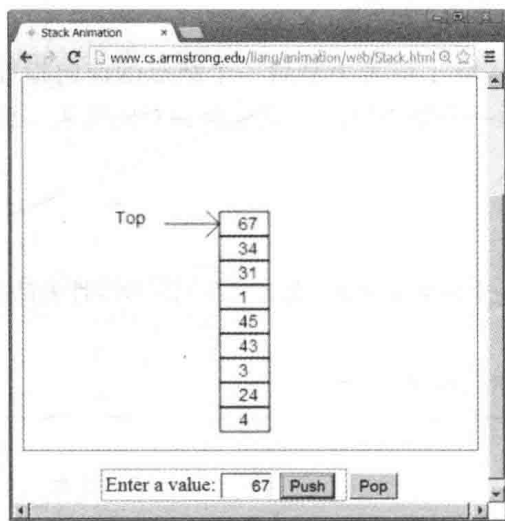
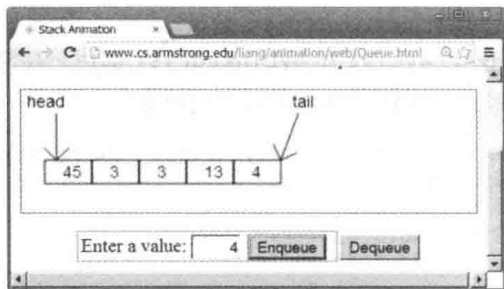


图 24-19 队列以先进先出的方式保存对象

教学注意：参见网址 www.cs.armstrong.edu/liang/animation/web/Stack.html 和 www.cs.armstrong.edu/liang/animation/web/Queue.html 来通过交互性演示查看栈和队列是如何工作的，如图 24-20 所示。



a) 栈动画



b) 队列动画

图 24-20 动画工具有助于了解栈和队列是如何工作的

由于栈只允许在栈顶进行插入与删除操作，所以用数组线性表来实现栈比用链表来实现效率更高。由于删除是在线性表的起始位置进行的，所以用链表实现队列比用数组线性表实现效率更高。本节将用数组线性表来实现栈，用链表来实现队列。

这里有两种办法可用来设计栈和队列的类。

- 使用继承：可以通过继承数组线性表类 `ArrayList` 来定义栈类，通过继承链表类 `LinkedList` 来定义队列类，如图 24-21a 所示。
- 使用组合：可以将数组线性表定义为栈类中的数据域，将链表定义为队列类中的数据域，如图 24-21b 所示。

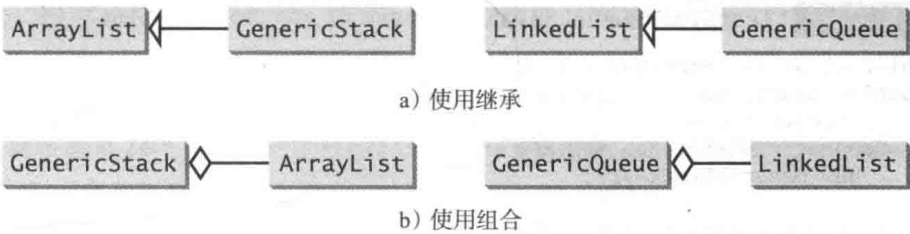


图 24-21 可以使用继承或组合实现 GenericStack 和 GenericQueue

这两种设计方法都是可行的，但是相比之下，组合可能更好一些，因为它可以定义一个全新的栈类和队列类，而不需要继承数组线性表类与链表类中不必要和不合适的方法。使用组合方式的栈类的实现已在程序清单 19-1 中给出。程序清单 24-7 使用组合方式实现队列类 GenericQueue。图 24-22 给出这个类的 UML。

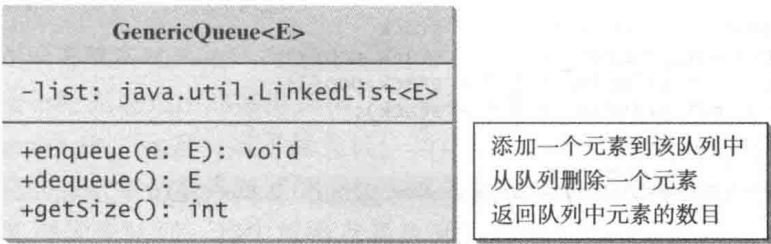


图 24-22 GenericQueue 使用链表来提供先进先出的数据结构

程序清单 24-7 GenericQueue.java

```
1 public class GenericQueue<E> {
2     private java.util.LinkedList<E> list
3       = new java.util.LinkedList<>();
4
5     public void enqueue(E e) {
6         list.addLast(e);
7     }
8
9     public E dequeue() {
10        return list.removeFirst();
11    }
12
13    public int getSize() {
14        return list.size();
15    }
16
17    @Override
18    public String toString() {
19        return "Queue: " + list.toString();
20    }
21 }
```

该程序创建一个链表来存储队列中的元素（第 2 ~ 3 行）。`enqueue(e)` 方法（第 5-7 行）将元素 `e` 添加到队列尾。`dequeue()` 方法（第 9 ~ 11 行）从队列头删除一个元素，并返回该元素。`getSize()` 方法（第 13 ~ 15 行）返回队列中元素的个数。

程序清单 24-8 给出一个使用 `GenericStack` 创建栈和使用 `GenericQueue` 创建队列的例子。它使用 `push(enqueue)` 方法向栈（或队列）中添加字符串，使用 `pop(dequeue)` 方法从栈（或队列）中删除字符串。

程序清单 24-8 TestStackQueue.java

```

1 public class TestStackQueue {
2     public static void main(String[] args) {
3         // Create a stack
4         GenericStack<String> stack =
5             new GenericStack<>();
6
7         // Add elements to the stack
8         stack.push("Tom"); // Push it to the stack
9         System.out.println("(1) " + stack);
10
11        stack.push("Susan"); // Push it to the the stack
12        System.out.println("(2) " + stack);
13
14        stack.push("Kim"); // Push it to the stack
15        stack.push("Michael"); // Push it to the stack
16        System.out.println("(3) " + stack);
17
18        // Remove elements from the stack
19        System.out.println("(4) " + stack.pop());
20        System.out.println("(5) " + stack.pop());
21        System.out.println("(6) " + stack);
22
23        // Create a queue
24        GenericQueue<String> queue = new GenericQueue<>();
25
26        // Add elements to the queue
27        queue.enqueue("Tom"); // Add it to the queue
28        System.out.println("(7) " + queue);
29
30        queue.enqueue("Susan"); // Add it to the queue
31        System.out.println("(8) " + queue);
32
33        queue.enqueue("Kim"); // Add it to the queue
34        queue.enqueue("Michael"); // Add it to the queue
35        System.out.println("(9) " + queue);
36
37        // Remove elements from the queue
38        System.out.println("(10) " + queue.dequeue());
39        System.out.println("(11) " + queue.dequeue());
40        System.out.println("(12) " + queue);
41    }
42 }

```

```

(1) stack: [Tom]
(2) stack: [Tom, Susan]
(3) stack: [Tom, Susan, Kim, Michael]
(4) Michael
(5) Kim
(6) stack: [Tom, Susan]
(7) Queue: [Tom]
(8) Queue: [Tom, Susan]
(9) Queue: [Tom, Susan, Kim, Michael]
(10) Tom
(11) Susan
(12) Queue: [Kim, Michael]

```

对一个栈来说，`push(e)` 方法将一个元素添加到栈顶，而 `pop()` 方法将栈顶元素从栈中删除并返回该元素。很容易得出，`push` 和 `pop` 方法的时间复杂度为 $O(1)$ 。

对一个队列来说，`enqueue(e)` 方法将一个元素添加到队列尾，而 `dequeue()` 方法从队列

头删除元素。很容易得出，enqueue 和 dequeue 方法的时间复杂度为 $O(1)$ 。

✓ 复习题

- 24.18 可以采用继承或组合来设计栈和队列的数据结构，试讨论这两种方法的优缺点。
- 24.19 如果程序清单 24-7 中第 2 ~ 3 行的 LinkedList 被替换为 ArrayList，enqueue 和 dequeue 方法的时间复杂度为多少？
- 24.20 下面代码的哪些行有错误？

```
1 List<String> list = new ArrayList<>();
2 list.add("Tom");
3 list = new LinkedList<>();
4 list.add("Tom");
5 list = new GenericStack<>();
6 list.add("Tom");
```

24.6 优先队列

🔑 要点提示：可以用堆实现优先队列。

普通的队列是一种先进先出的数据结构，元素在队列尾追加，而从队列头删除。在优先队列（priority queue）中，元素被赋予优先级。当访问元素时，具有最高优先级的元素最先删除。例如，医院的急救室为病人赋予优先级，具有最高优先级的病人最先得到治疗。

可以使用堆实现优先队列，其中根结点是队列中具有最高优先级的对象。本书在 23.6 节中介绍过堆。优先队列的类图如图 24-23 所示，它的实现在程序清单 24-9 中给出。

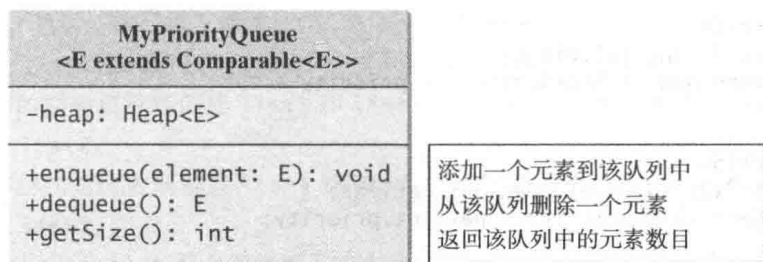


图 24-23 MyPriorityQueue 使用堆提供一种最高进先出（largest-in, first-Out）的数据结构

程序清单 24-9 MyPriorityQueue.java

```
1 public class MyPriorityQueue<E extends Comparable<E>> {
2     private Heap<E> heap = new Heap<>();
3
4     public void enqueue(E newObject) {
5         heap.add(newObject);
6     }
7
8     public E dequeue() {
9         return heap.remove();
10    }
11
12    public int getSize() {
13        return heap.getSize();
14    }
15 }
```

程序清单 24-10 给出一个对病人使用优先队列的例子。Patient 类在第 19 ~ 37 行定义。第 3 ~ 6 行创建带相关优先值的 4 个病人。第 8 行创建一个优先队列。病人在第 10 ~ 13 行

加入队列。第 16 行让一个病人移出队列。

程序清单 24-10 TestPriorityQueue.java

```

1 public class TestPriorityQueue {
2     public static void main(String[] args) {
3         Patient patient1 = new Patient("John", 2);
4         Patient patient2 = new Patient("Jim", 1);
5         Patient patient3 = new Patient("Tim", 5);
6         Patient patient4 = new Patient("Cindy", 7);
7
8         MyPriorityQueue<Patient> priorityQueue
9             = new MyPriorityQueue<>();
10        priorityQueue.enqueue(patient1);
11        priorityQueue.enqueue(patient2);
12        priorityQueue.enqueue(patient3);
13        priorityQueue.enqueue(patient4);
14
15        while (priorityQueue.getSize() > 0)
16            System.out.print(priorityQueue.dequeue() + " ");
17    }
18
19    static class Patient implements Comparable<Patient> {
20        private String name;
21        private int priority;
22
23        public Patient(String name, int priority) {
24            this.name = name;
25            this.priority = priority;
26        }
27
28        @Override
29        public String toString() {
30            return name + "(priority:" + priority + ")";
31        }
32
33        @Override
34        public int compareTo(Patient patient) {
35            return this.priority - patient.priority;
36        }
37    }
38 }

```

Cindy(priority:7) Tim(priority:5) John(priority:2) Jim(priority:1)

✓ 复习题

24.21 什么是优先队列？

24.22 MyPriorityQueue 中的 enqueue、dequeue 以及 getSize 方法的时间复杂度为多少？

24.23 下面语句哪些有错误？

```

1 MyPriorityQueue<Object> q1 = new MyPriorityQueue<>();
2 MyPriorityQueue<Number> q2 = new MyPriorityQueue<>();
3 MyPriorityQueue<Integer> q3 = new MyPriorityQueue<>();
4 MyPriorityQueue<Date> q4 = new MyPriorityQueue<>();
5 MyPriorityQueue<String> q5 = new MyPriorityQueue<>();

```

本章小结

1. 本章学习了如何实现数组线性表、链表、栈以及队列。
2. 定义一个数据结构本质上是定义一个类。为数据结构定义的类应该使用数据域来存储数据，并提供方法来支持诸如插入和删除等操作。

3. 创建一个数据结构是从该类创建一个实例。这样就可以将方法应用在实例上来处理数据结构，比如插入一个元素到数据结构中，或者从数据结构中删除一个元素。
4. 本章学习了如何采用堆来实现一个优先队列。

测试题

回答位于网址 www.cs.armstrong.edu/liang/intro10e/test.html 的本章测试题。

编程练习题

- 24.1 (在 `MyList` 中添加集合操作) 在 `MyList` 中定义下列方法，并在 `MyAbstractList` 中实现下列方法：

```
/** Adds the elements in otherList to this list.
 * Returns true if this list changed as a result of the call */
public boolean addAll(MyList<E> otherList);

/** Removes all the elements in otherList from this list
 * Returns true if this list changed as a result of the call */
public boolean removeAll(MyList<E> otherList);

/** Retains the elements in this list that are also in otherList
 * Returns true if this list changed as a result of the call */
public boolean retainAll(MyList<E> otherList);
```

编写一个测试程序，创建两个 `MyArrayList` 对象 `list1` 和 `list2`，初始值分别为 {"Tom", "George", "Peter", "Jean", "Jane"} 和 {"Tom", "George", "Michael", "Michelle", "Daniel"}。然后，执行以下操作：

- 调用方法 `list1.addAll(list2)`，并显示 `list1` 和 `list2`。
 - 采用同样的初始值重新创建 `list1` 和 `list2`，然后调用 `list1.removeAll(list2)`，并显示 `list1` 和 `list2`。
 - 采用同样的初始值重新创建 `list1` 和 `list2`，然后调用 `list1.retainAll(list2)`，并显示 `list1` 和 `list2`。
- *24.2 (实现 `MyLinkedList`) 本书中省略了下述方法的实现，请实现它们：`contains(E e)`、`get(int index)`、`indexOf(E e)`、`lastIndexOf(E e)` 和 `set(int index, E e)`。
- *24.3 (实现双向链表) 程序清单 24-6 中使用的 `MyLinkedList` 类创建了一个单向链表，它只能单向遍历线性表。修改 `Node` 类，添加一个名为 `previous` 的数据域，让它指向链表中的前一个结点，如下所示：

```
public class Node<E> {
    E element;
    Node<E> next;
    Node<E> previous;

    public Node(E e) {
        element = e;
    }
}
```

实现一个名为 `TwoWayLinkedList` 的新类，使用双向链表来存储元素。课本中的 `MyLinkedList` 类继承自 `MyAbstractList`。定义 `TwoWayLinkedList` 继承 `java.util.AbstractSequentialList` 类。不光要实现 `listIterator()` 和 `listIterator(int index)` 方法，还要实现定义在 `MyLinkedList` 中包括的所有方法。都返回一个 `java.util.ListIterator<E>` 类型的实例。前者指向线性表的头部，后者指向指定下标的元素。

24.4 (使用 GenericStack 类) 编写一个程序, 以降序显示前 50 个素数。使用栈存储素数。

24.5 (利用继承关系实现 GenericQueue) 24.5 节使用组合关系实现了 GenericQueue。继承 java.util.LinkedList 类, 创建一个新的队列类。

*24.6 (使用 Comparator 实现泛型 PriorityQueue) 修改程序清单 24-9 中的 MyPriorityQueue, 使用一个泛型参数来比较对象。如下定义一个使用 Comparator 作为参数的新的构造方法:

```
PriorityQueue(Comparator<? super E> comparator)
```

**24.7 (动画: 链表) 编写一个程序, 用动画实现链表的查找、插入和删除, 如图 24-1b 所示。按钮 Search 用来查找一个指定的值是否在链表中; 按钮 Delete 用来从链表中删除一个特定值; 按钮 Insert 用来在链表的特定下标处插入一个值, 如果没有指定下标, 则添加到链表的末尾。

*24.8 (动画: 数组线性表) 编写一个程序, 用动画实现数组线性表的查找、插入和删除, 如图 24-1a 所示。按钮 Search 用来查找一个指定的值是否在线性表中; 按钮 Delete 用来从线性表中删除一个特定值; 按钮 Insert 用来在链表的特定下标处插入一个值, 如果没有指定下标, 则添加到链表的末尾。

*24.9 (动画: 慢动作显示数组线性表) 改进前面编程练习题, 通过慢动作显示插入和删除操作, 如网址 <http://www.cs.armstrong.edu/liang/animation/ArrayListAnimationInSlowMotion.html> 所示。

*24.10 (动画: 栈) 编写一个程序, 用动画实现栈的压入和弹出, 如图 24-20a 所示。

*24.11 (动画: 双向链表) 编写一个程序, 用动画实现双向链表的查找、插入和删除, 如图 24-24 所示。按钮 Search 用来查找一个指定的值是否在线性表中; 按钮 Delete 用来从线性表中删除一个特定值; 按钮 Insert 用来在链表的特定下标处插入一个值, 如果没有指定下标, 则添加到链表的末尾。同时, 添加两个名为 Forward Traversal 和 Backward Traversal 的按钮, 用于采用遍历器分别以向前和往后的方式来显示元素。

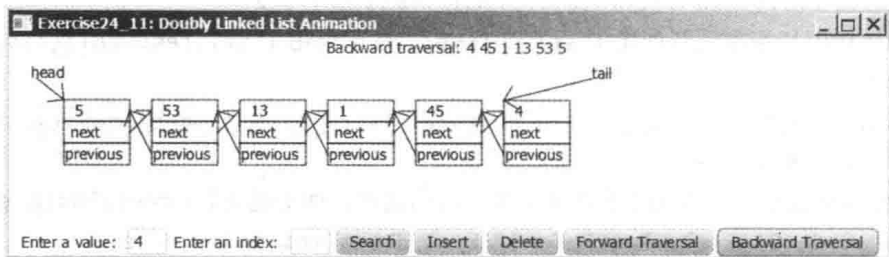


图 24-24 程序实现双向链表的运行动画

*24.12 (动画: 队列) 编写一个程序, 用动画实现队列的 enqueue 和 dequeue 操作, 如图 24-20b 所示。

*24.13 (斐波那契数遍历器) 定义一个名为 FibonacciIterator 的遍历器, 用于遍历斐波那契数字。构造方法带有一个参数, 用于指定斐波那契数字的上限。比如, new FibonacciIterator (23302) 创建一个遍历器, 可以用于遍历小于或者等于 23302 的斐波那契数。编写一个测试程序, 使用该遍历器显示所有小于或者等于 100000 的斐波那契数。

*24.14 (素数遍历器) 定义一个名为 PrimeIterator 的遍历器, 用于遍历素数。构造方法带有一个参数, 用于指定斐波那契数字的上限。比如, new PrimeIterator (23302) 创建一个遍历器, 可以用于遍历小于或者等于 23302 的素数。编写一个测试程序, 使用该遍历器显示所有小于或者等于 100000 的素数。

**24.15 (测试 MyArrayList) 设计和编写一个完整的测试程序, 用于测试程序清单 24-3 中的 MyArrayList 类是否符合所有的要求。

**24.16 (测试 MyLinkedList) 设计和编写一个完整的测试程序, 用于测试程序清单 24-6 中的 MyLinkedList 类是否符合所有的要求。

二叉查找树

教学目标

- 设计并实现二叉查找树（25.2 节）。
- 使用链式数据结构表示二叉树（25.2.1 节）。
- 在二叉查找树中查找元素（25.2.2 节）。
- 在二叉查找树中插入元素（25.2.3 节）。
- 遍历二叉树中的元素（25.2.4 节）。
- 设计和实现 Tree 接口、AbstractTree 类以及 BST 类（25.2.5 节）。
- 从二叉查找树中删除元素（25.3 节）。
- 图形化显示二叉树（25.4 节）。
- 创建迭代器来遍历二叉树（25.5 节）。
- 使用二叉树实现用于压缩数据的霍夫曼编码（25.6 节）。

25.1 引言

要点提示：树是一种典型的数据结构，具有很多重要的应用。

树（tree）提供了一种层次组织结构，数据可以存储在树中的每个结点内。本章将介绍二叉查找树。你将学到如何构建二叉查找树，如何查找元素、插入元素、删除元素，以及在二叉查找树中遍历元素。你还将学到如何定义和实现一个自定义的数据结构，实现二叉查找树。

25.2 二叉查找树

要点提示：二叉查找树可以用链接结构实现。

回顾一下，线性表、栈和队列都是由一系列元素构成的线性结构。二叉树（binary tree）是一种层次结构，它要么是空集，要么是由一个称为根（root）的元素和两棵不同的二叉树组成的，这两棵二叉树分别称为左子树（left subtree）和右子树（right subtree）。允许这两棵子树中的一棵或者两棵为空。二叉树的示例如图 25-1 所示。

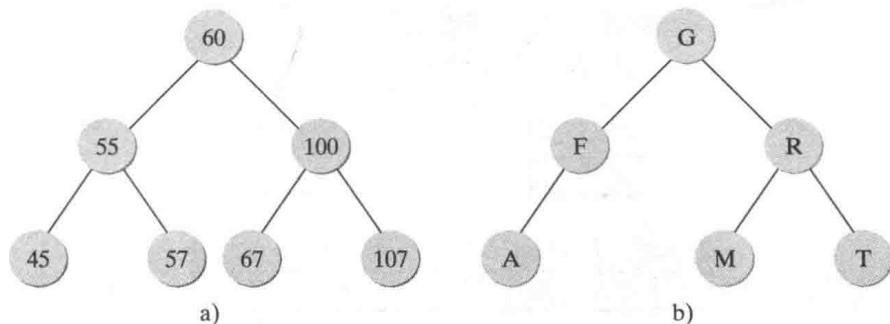


图 25-1 二叉树的每个结点有 0 棵、1 棵或 2 棵子树

一条路径的长度 (length) 是指在该条路径上的边的个数。一个结点的深度 (depth) 是指从根结点到该结点的路径长度。有时候, 我们将一棵树中具有某个给定深度的所有结点的集合称为该树的层 (level)。兄弟结点 (sibling) 是共享同一父结点的结点。一个结点的左 (右) 子树的根结点称为这个结点的左 (右) 子结点 (left (right) child)。没有子结点的结点称为叶结点 (leaf)。非空树的高度为从根结点到最远的叶结点的路径长度。只有一个结点的树高度为 0。习惯上, 将空树的高度定为 -1。考虑图 25-1a 中的树。从结点 60 到 45 的路径的长度为 2。结点 60 的深度为 0, 结点 55 的深度为 1, 而结点 45 的深度为 2。这棵树的高度为 2。结点 45 和 57 是兄弟结点。结点 45、57、67 和 107 位于同一层。

一种称为二叉查找树 (binary search tree, BST) 的特殊类型的二叉树非常有用。二叉查找树 (没有重复元素) 的特征是: 对于树中的每一个结点, 它的左子树中结点的值都小于该结点的值, 而它的右子树中结点的值都大于该结点的值。图 25-1 中的二叉树都是二叉查找树。

教学注意: 参见链接 www.cs.armstrong.edu/liang/animation/web/BST.html 查看 BST 运行机制的在线交互式演示, 如图 25-2 所示。

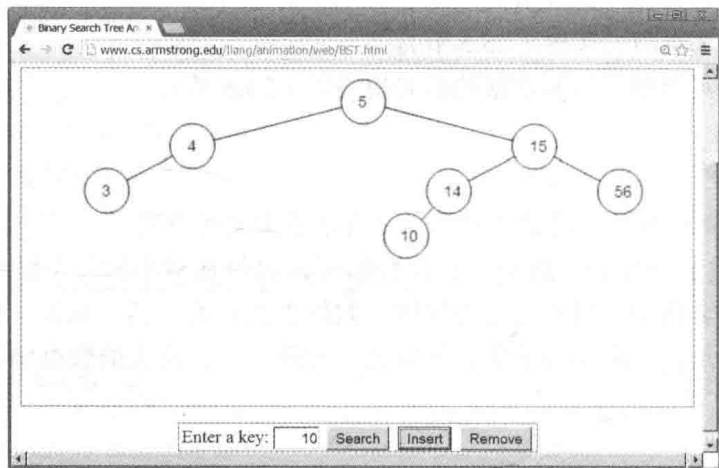


图 25-2 动画工具可以让你插入、删除和查找元素

25.2.1 表示二叉查找树

可以使用一个链式结点的集合来表示二叉树。每个结点都包含一个数值和两个称为 left 和 right 的链接, 分别指向左孩子和右孩子, 如图 25-3 所示。

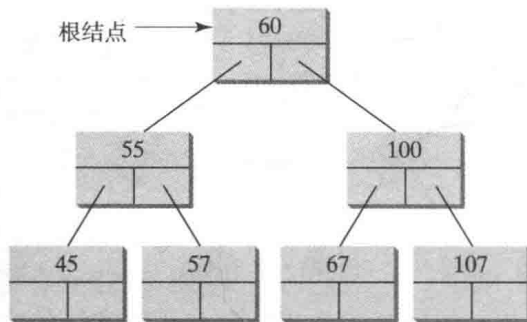


图 25-3 使用链式结点的集合表示二叉树

结点可以定义为一个类，如下所示：

```
class TreeNode<E> {  
    protected E element;  
    protected TreeNode<E> left;  
    protected TreeNode<E> right;  
  
    public TreeNode(E e) {  
        element = e;  
    }  
}
```

变量 `root` 指向树的根结点。如果树为空，那么 `root` 的值为 `null`。下面的代码创建了如图 25-3 所示的树的前三个结点：

```
// Create the root node  
TreeNode<Integer> root = new TreeNode<>(60);  
  
// Create the left child node  
root.left = new TreeNode<>(55);  
  
// Create the right child node  
root.right = new TreeNode<>(100);
```

25.2.2 查找一个元素

要在二叉查找树中查找一个元素，可从根结点开始向下扫描，直到找到一个匹配元素，或者达到一棵空子树为止。该算法在程序清单 25-1 中描述。让 `current` 指向根结点（第 2 行），重复下面的步骤直到 `current` 为 `null`（第 4 行）或者元素匹配 `current.element`（第 12 行）。

- 如果 `element` 小于 `current.element`，就将 `current.left` 赋给 `current`（第 6 行）。
- 如果 `element` 大于 `current.element`，就将 `current.right` 赋给 `current`（第 9 行）。
- 如果 `element` 等于 `current.element`，就返回 `true`（第 12 行）。

如果 `current` 为 `null`，那么子树为空且该元素不在这棵树中（第 14 行）。

程序清单 25-1 在 BST 中查找一个元素

```
1 public boolean search(E element) {  
2     TreeNode<E> current = root; // Start from the root  
3  
4     while (current != null)  
5         if (element < current.element) {  
6             current = current.left; // Go left  
7         }  
8         else if (element > current.element) {  
9             current = current.right; // Go right  
10        }  
11        else // Element matches current.element  
12            return true; // Element is found  
13  
14    return false; // Element is not in the tree  
15 }
```

25.2.3 在 BST 中插入一个元素

为了在 BST 中插入一个元素，需要确定在树中插入元素的位置。关键思路是确定新结点的父结点所在的位置。程序清单 25-2 给出该算法。

程序清单 25-2 在 BST 中插入一个元素

```

1  boolean insert(E e) {
2      if (tree is empty)
3          // Create the node for e as the root;
4      else {
5          // Locate the parent node
6          parent = current = root;
7          while (current != null)
8              if (e < the value in current.element) {
9                  parent = current; // Keep the parent
10                 current = current.left; // Go left
11             }
12             else if (e > the value in current.element) {
13                 parent = current; // Keep the parent
14                 current = current.right; // Go right
15             }
16             else
17                 return false; // Duplicate node not inserted
18
19         // Create a new node for e and attach it to parent
20
21         return true; // Element inserted
22     }
23 }

```

如果这棵树是空的, 就使用新元素创建一个根结点 (第 2 ~ 3 行); 否则, 寻找新元素结点的父结点的位置 (第 6 ~ 17 行)。为该元素创建一个新结点, 然后将该结点链接到它的父结点上。如果新元素的值小于父元素的值, 则将新元素的结点设置为父结点的左子结点; 如果新元素的值大于父元素的值, 则将新元素的结点设置为父结点的右子结点。

例如, 要将数据 101 插入图 25-3 所示的树中, 在算法中的 while 循环结束之后, parent 指向存储数据 107 的结点, 如图 25-4a 所示。存储数据 101 的新结点将成为父结点的左子结点。要将数据 59 插入树中, 在算法中的 while 循环结束之后, 父结点指向存储数据 57 的结点, 如图 25-4b 所示。存储数据 59 的新结点成为父结点的右子结点。

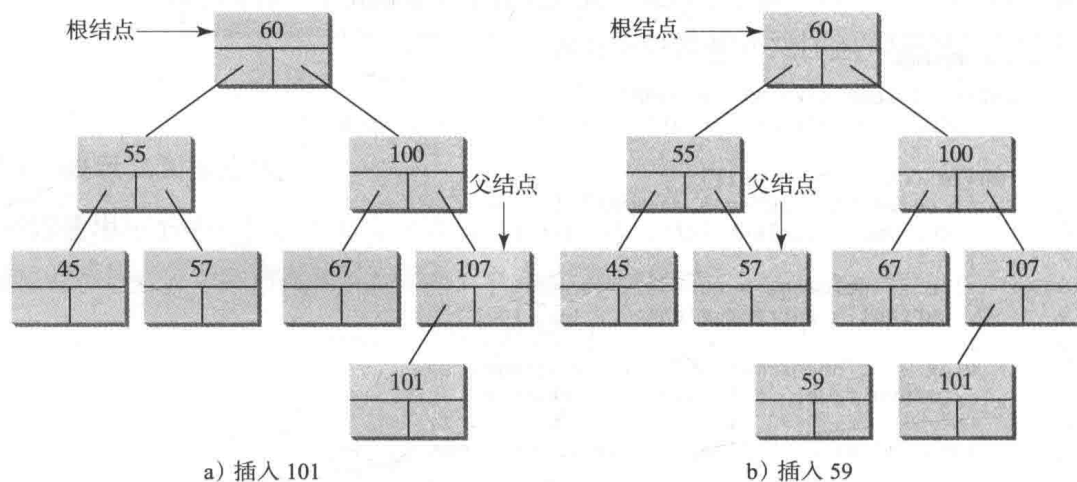


图 25-4 在树中插入两个新元素

25.2.4 树的遍历

树的遍历 (tree traversal) 就是访问树中每个结点一次且只有一次的过程。遍历树的方法有很多种。本节将介绍中序 (inorder)、前序 (preorder)、后序 (postorder)、深度优先 (depth-

first) 和广度优先 (breadth-first) 等遍历方法。

中序遍历 (inorder traversal) 法, 首先递归地访问当前结点的左子树, 然后访问当前结点, 最后递归地访问该结点的右子树。中序遍历法以递增顺序显示 BST 中的所有结点。

后序遍历 (postorder traversal) 法, 首先递归地访问当前结点的左子树, 然后递归地访问该结点的右子树, 最后访问该结点本身。后序遍历的一个应用就是找出一个文件系统中目录的个数。如图 25-5 所示, 每个目录都是一个内部结点, 而每个文件都是叶结点。可以使用后序遍历法, 在找出根目录的大小之前得到每个文件和子目录的大小。

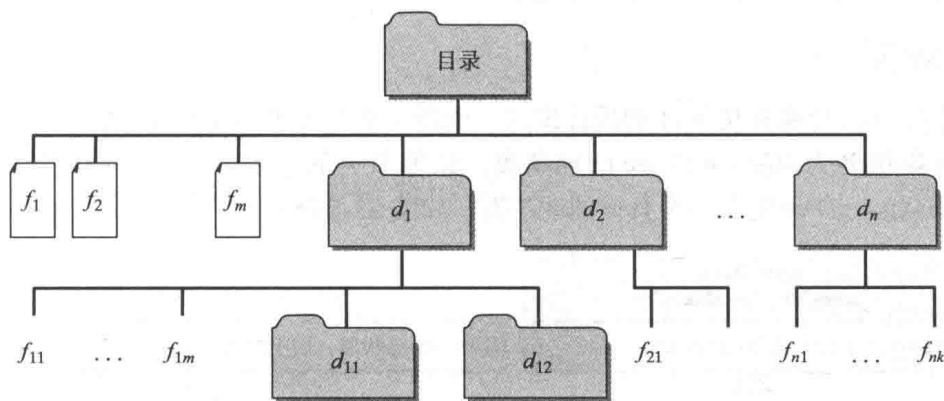


图 25-5 一个目录包括文件和子目录

前序遍历 (preorder traversal) 法, 首先访问当前结点, 然后递归地访问该结点的左子树, 最后递归地访问该结点的右子树。深度优先遍历法与前序遍历法相同。前序遍历的一个应用就是打印一个结构性文档。如图 25-6 所示, 可以使用前序遍历法打印本书的目录。

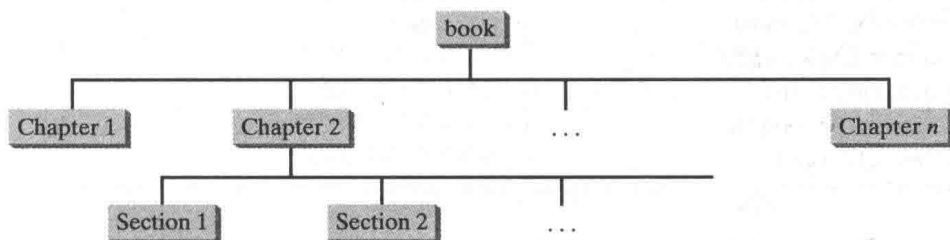


图 25-6 树可以用来表示一个结构性文档，例如一本书、一章和一节

{ } 注意: 可以采用前序插入元素的方法重构一棵二叉查找树。重构的树为原始的二叉查找树保留了父结点和子结点的关系。

广度优先遍历法逐层访问树中的结点。首先访问根结点, 然后从左往右访问根结点的所有子结点, 再从左往右访问根结点的所有孙子结点, 以此类推。

例如, 对于图 25-4b 中的树, 它的中序遍历为

45 55 57 59 60 67 100 101 107

它的后序遍历为

45 59 57 55 67 101 107 100 60

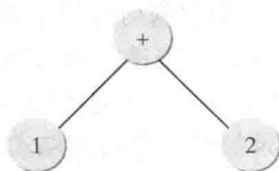
它的前序遍历为

60 55 45 57 59 100 67 107 101

它的广度优先遍历为

60 55 100 45 57 67 107 59 101

可以使用下面的树来帮助记忆中序、后序以及前序：



中序是 1 + 2，后序是 1 2 +，前序是 + 1 2。

25.2.5 BST 类

我们遵循 Java 合集框架 API 的设计模式，使用一个名为 `Tree` 的接口来定义树的所有常用操作，并提供名为 `AbstractTree` 的抽象类，该抽象类部分地实现了 `Tree`，如图 25-7 所示。继承 `AbstractTree` 定义一个具体的 BST 类，如图 25-8 所示。

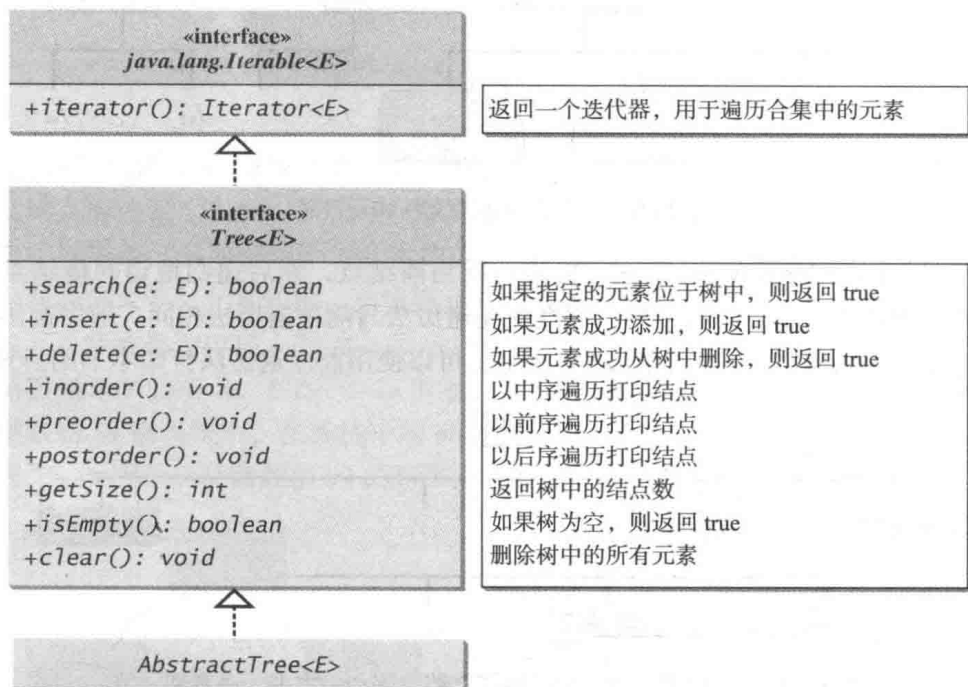


图 25-7 Tree 接口定义树的常用操作，而 `AbstractTree` 类部分地实现了 `Tree`

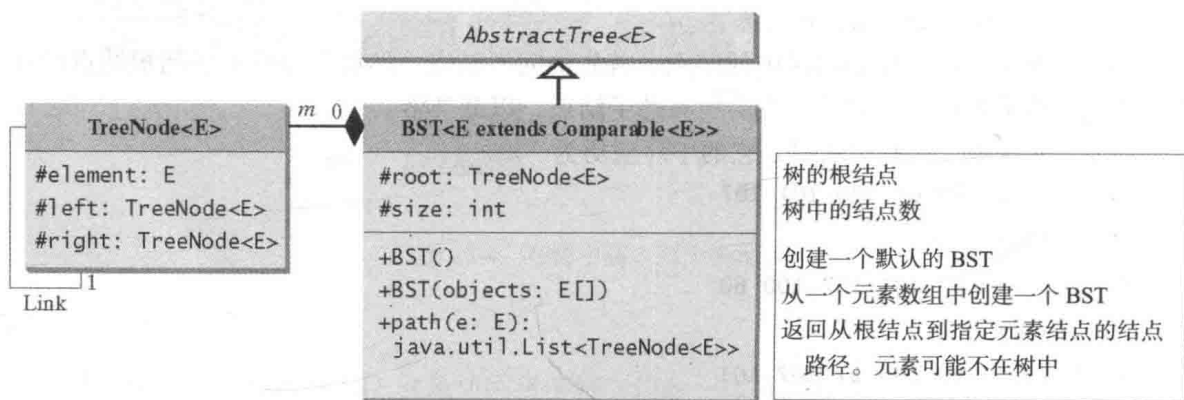


图 25-8 BST 类定义了一个具体的 BST

程序清单 25-3、程序清单 25-4 和程序清单 25-5 分别给出 Tree、AbstractTree 和 BST 的实现。

程序清单 25-3 Tree.java

```
1 public interface Tree<E> extends Iterable<E> {
2     /** Return true if the element is in the tree */
3     public boolean search(E e);
4
5     /** Insert element e into the binary search tree.
6      * Return true if the element is inserted successfully. */
7     public boolean insert(E e);
8
9     /** Delete the specified element from the tree.
10      * Return true if the element is deleted successfully. */
11     public boolean delete(E e);
12
13     /** Inorder traversal from the root*/
14     public void inorder();
15
16     /** Postorder traversal from the root */
17     public void postorder();
18
19     /** Preorder traversal from the root */
20     public void preorder();
21
22     /** Get the number of nodes in the tree */
23     public int getSize();
24
25     /** Return true if the tree is empty */
26     public boolean isEmpty();
27 }
```

程序清单 25-4 AbstractTree.java

```
1 public abstract class AbstractTree<E>
2     implements Tree<E> {
3     @Override /** Inorder traversal from the root*/
4     public void inorder() {
5     }
6
7     @Override /** Postorder traversal from the root */
8     public void postorder() {
9     }
10
11     @Override /** Preorder traversal from the root */
12     public void preorder() {
13     }
14
15     @Override /** Return true if the tree is empty */
16     public boolean isEmpty() {
17         return getSize() == 0;
18     }
19 }
```

程序清单 25-5 BST.java

```
1 public class BST<E> extends Comparable<E>>
2     extends AbstractTree<E> {
3     protected TreeNode<E> root;
4     protected int size = 0;
5
6     /** Create a default binary search tree */
```

```

7  public BST() {
8  }
9
10 /** Create a binary search tree from an array of objects */
11 public BST(E[] objects) {
12     for (int i = 0; i < objects.length; i++)
13         insert(objects[i]);
14 }
15
16 @Override /** Return true if the element is in the tree */
17 public boolean search(E e) {
18     TreeNode<E> current = root; // Start from the root
19
20     while (current != null) {
21         if (e.compareTo(current.element) < 0) {
22             current = current.left;
23         }
24         else if (e.compareTo(current.element) > 0) {
25             current = current.right;
26         }
27         else // element matches current.element
28             return true; // Element is found
29     }
30
31     return false;
32 }
33
34 @Override /** Insert element e into the binary search tree.
35  * Return true if the element is inserted successfully. */
36 public boolean insert(E e) {
37     if (root == null)
38         root = createNewNode(e); // Create a new root
39     else {
40         // Locate the parent node
41         TreeNode<E> parent = null;
42         TreeNode<E> current = root;
43         while (current != null)
44             if (e.compareTo(current.element) < 0) {
45                 parent = current;
46                 current = current.left;
47             }
48             else if (e.compareTo(current.element) > 0) {
49                 parent = current;
50                 current = current.right;
51             }
52             else
53                 return false; // Duplicate node not inserted
54
55         // Create the new node and attach it to the parent node
56         if (e.compareTo(parent.element) < 0)
57             parent.left = createNewNode(e);
58         else
59             parent.right = createNewNode(e);
60     }
61
62     size++;
63     return true; // Element inserted successfully
64 }
65
66 protected TreeNode<E> createNewNode(E e) {
67     return new TreeNode<>(e);
68 }
69
70 @Override /** Inorder traversal from the root */

```

```

71 public void inorder() {
72     inorder(root);
73 }
74
75 /** Inorder traversal from a subtree */
76 protected void inorder(TreeNode<E> root) {
77     if (root == null) return;
78     inorder(root.left);
79     System.out.print(root.element + " ");
80     inorder(root.right);
81 }
82
83 @Override /** Postorder traversal from the root */
84 public void postorder() {
85     postorder(root);
86 }
87
88 /** Preorder traversal from a subtree */
89 protected void preorder(TreeNode<E> root) {
90     if (root == null) return;
91     postorder(root.left);
92     postorder(root.right);
93     System.out.print(root.element + " ");
94 }
95
96 @Override /** Preorder traversal from the root */
97 public void preorder() {
98     preorder(root);
99 }
100
101 /** Postorder traversal from a subtree */
102 protected void postorder(TreeNode<E> root) {
103     if (root == null) return;
104     System.out.print(root.element + " ");
105     preorder(root.left);
106     preorder(root.right);
107 }
108
109 /** This inner class is static, because it does not access
110     any instance members defined in its outer class */
111 public static class TreeNode<E> extends Comparable<E>> {
112     protected E element;
113     protected TreeNode<E> left;
114     protected TreeNode<E> right;
115
116     public TreeNode(E e) {
117         element = e;
118     }
119 }
120
121 @Override /** Get the number of nodes in the tree */
122 public int getSize() {
123     return size;
124 }
125
126 /** Returns the root of the tree */
127 public TreeNode<E> getRoot() {
128     return root;
129 }
130
131 /** Returns a path from the root leading to the specified element */
132 public java.util.ArrayList<TreeNode<E>> path(E e) {
133     java.util.ArrayList<TreeNode<E>> list =
134         new java.util.ArrayList<>();

```

```

135     TreeNode<E> current = root; // Start from the root
136
137     while (current != null) {
138         list.add(current); // Add the node to the list
139         if (e.compareTo(current.element) < 0) {
140             current = current.left;
141         }
142         else if (e.compareTo(current.element) > 0) {
143             current = current.right;
144         }
145         else
146             break;
147     }
148
149     return list; // Return an array list of nodes
150 }
151
152 @Override /** Delete an element from the binary search tree.
153  * Return true if the element is deleted successfully.
154  * Return false if the element is not in the tree. */
155 public boolean delete(E e) {
156     // Locate the node to be deleted and also locate its parent node
157     TreeNode<E> parent = null;
158     TreeNode<E> current = root;
159     while (current != null) {
160         if (e.compareTo(current.element) < 0) {
161             parent = current;
162             current = current.left;
163         }
164         else if (e.compareTo(current.element) > 0) {
165             parent = current;
166             current = current.right;
167         }
168         else
169             break; // Element is in the tree pointed at by current
170     }
171
172     if (current == null)
173         return false; // Element is not in the tree
174
175     // Case 1: current has no left child
176     if (current.left == null) {
177         // Connect the parent with the right child of the current node
178         if (parent == null) {
179             root = current.right;
180         }
181         else {
182             if (e.compareTo(parent.element) < 0)
183                 parent.left = current.right;
184             else
185                 parent.right = current.right;
186         }
187     }
188     else {
189         // Case 2: The current node has a left child.
190         // Locate the rightmost node in the left subtree of
191         // the current node and also its parent.
192         TreeNode<E> parentOfRightMost = current;
193         TreeNode<E> rightMost = current.left;
194
195         while (rightMost.right != null) {
196             parentOfRightMost = rightMost;
197             rightMost = rightMost.right; // Keep going to the right
198         }

```



```

199
200     // Replace the element in current by the element in rightMost
201     current.element = rightMost.element;
202
203     // Eliminate rightmost node
204     if (parentOfRightMost.right == rightMost)
205         parentOfRightMost.right = rightMost.left;
206     else
207         // Special case: parentOfRightMost == current
208         parentOfRightMost.left = rightMost.left;
209 }
210
211 size--;
212 return true; // Element deleted successfully
213 }
214
215 @Override /** Obtain an iterator. Use inorder. */
216 public java.util.Iterator<E> iterator() {
217     return new InorderIterator();
218 }
219
220 // Inner class InorderIterator
221 private class InorderIterator implements java.util.Iterator<E> {
222     // Store the elements in a list
223     private java.util.ArrayList<E> list =
224         new java.util.ArrayList<>();
225     private int current = 0; // Point to the current element in list
226
227     public InorderIterator() {
228         inorder(); // Traverse binary tree and store elements in list
229     }
230
231     /** Inorder traversal from the root */
232     private void inorder() {
233         inorder(root);
234     }
235
236     /** Inorder traversal from a subtree */
237     private void inorder(TreeNode<E> root) {
238         if (root == null) return;
239         inorder(root.left);
240         list.add(root.element);
241         inorder(root.right);
242     }
243
244     @Override /** More elements for traversing? */
245     public boolean hasNext() {
246         if (current < list.size())
247             return true;
248
249         return false;
250     }
251
252     @Override /** Get the current element and move to the next */
253     public E next() {
254         return list.get(current++);
255     }
256
257     @Override /** Remove the current element */
258     public void remove() {
259         delete(list.get(current)); // Delete the current element
260         list.clear(); // Clear the list
261         inorder(); // Rebuild the list
262     }

```

```

263     }
264
265     /** Remove all elements from the tree */
266     public void clear() {
267         root = null;
268         size = 0;
269     }
270 }

```

方法 `insert(E e)` (第 36 ~ 64 行) 为元素 `e` 创建一个结点, 并将它插入树中。如果树是空的, 则该结点就成为根结点; 否则, 该方法为这个结点寻找一个能够保持树的顺序的父结点。如果此元素已经在树中, 则该方法返回 `false`; 否则, 返回 `true`。

方法 `inorder()` (第 71 ~ 81 行) 调用 `inorder(root)` 遍历整棵树。`inorder(TreeNode root)` 方法从指定的根结点遍历树。它是一个递归方法, 先递归地遍历左子树, 然后遍历根结点, 最后遍历右子树。当树为空时, 遍历结束。

方法 `preorder()` (第 84 ~ 94 行) 与 `postorder()` (第 97 ~ 107 行) 的实现很类似, 都是使用递归来实现的。

方法 `path(E e)` (第 132 ~ 150 行) 以数组线性表返回结点的路径, 即从根结点开始到该元素所在的结点。元素可能不在树中。例如, 在图 25-4a 中, `path(45)` 包含元素 60、55 和 45 的结点, 而 `path(58)` 包含元素 60、55 和 57 的结点。

在 25.3 节和 25.5 节中讨论 `delete()` 和 `iterator()` 的实现 (第 155 ~ 269 行)。

程序清单 25-6 给出一个例子, 使用 `BST` (第 4 行) 创建一棵二叉查找树。程序向树中添加一些字符串 (第 5 ~ 11 行), 然后对该树进行中序、后序和前序遍历 (第 14 ~ 20 行), 查找一个元素 (第 24 行), 以及获取一个从包含 `Peter` 的结点到根结点的路径 (第 28 ~ 31 行)。

程序清单 25-6 TestBST.java

```

1  public class TestBST {
2      public static void main(String[] args) {
3          // Create a BST
4          BST<String> tree = new BST<>();
5          tree.insert("George");
6          tree.insert("Michael");
7          tree.insert("Tom");
8          tree.insert("Adam");
9          tree.insert("Jones");
10         tree.insert("Peter");
11         tree.insert("Daniel");
12
13         // Traverse tree
14         System.out.print("Inorder (sorted): ");
15         tree.inorder();
16         System.out.print("\nPostorder: ");
17         tree.postorder();
18         System.out.print("\nPreorder: ");
19         tree.preorder();
20         System.out.print("\nThe number of nodes is " + tree.getSize());
21
22         // Search for an element
23         System.out.print("\nIs Peter in the tree? " +
24             tree.search("Peter"));
25
26         // Get a path from the root to Peter
27         System.out.print("\nA path from the root to Peter is: ");
28         java.util.ArrayList<BST.TreeNode<String>> path

```

```

29     = tree.path("Peter");
30     for (int i = 0; path != null && i < path.size(); i++)
31         System.out.print(path.get(i).element + " ");
32
33     Integer[] numbers = {2, 4, 3, 1, 8, 5, 6, 7};
34     BST<Integer> intTree = new BST<>(numbers);
35     System.out.print("\nInorder (sorted): ");
36     intTree.inorder();
37 }
38 }
    
```

```

Inorder (sorted): Adam Daniel George Jones Michael Peter Tom
Postorder: Daniel Adam Jones Peter Tom Michael George
Preorder: George Adam Daniel Michael Jones Tom Peter
The number of nodes is 7
Is Peter in the tree? true
A path from the root to Peter is: George Michael Tom Peter
Inorder (sorted): 1 2 3 4 5 6 7 8
    
```

程序在第 30 行检查 `path!=null`，以确保在调用 `path.get(i)` 之前路径不是 `null`。这是一个防御性编程的例子，以避免潜在运行时错误。

程序创建另一棵树来存储 `int` 值（第 34 行）。在树中插入所有的元素后，该树应该如图 25-9 所示。

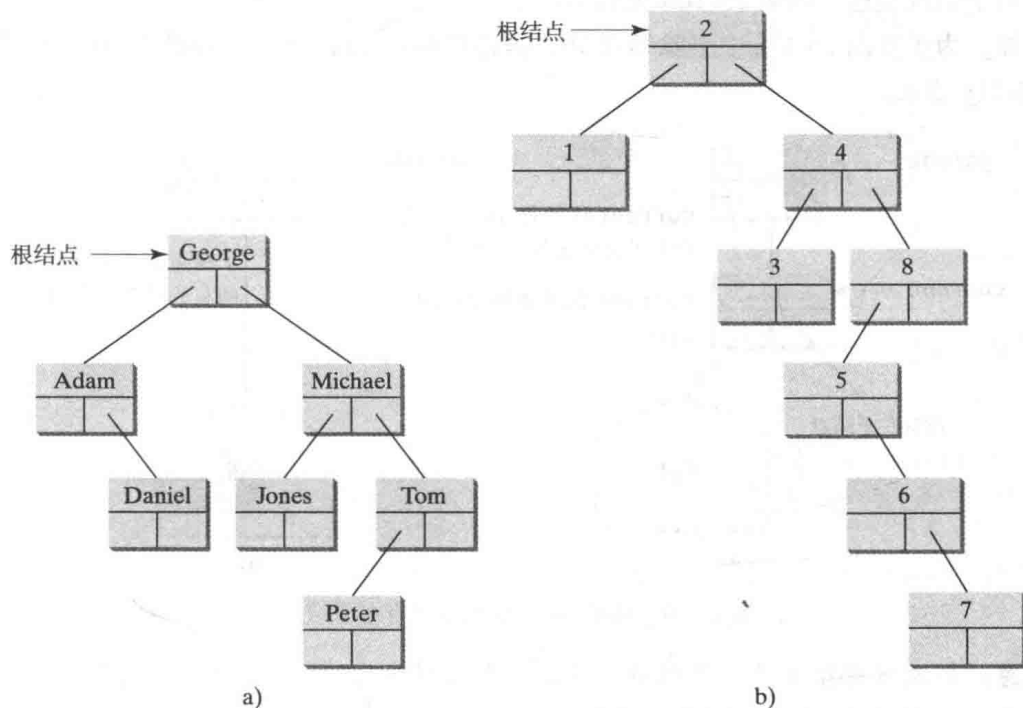


图 25-9 这里显示程序清单 25-6 中创建的几个 BST

如果元素的插入顺序不同（例如，Daniel、Adam、Jones、Peter、Tom、Michael、George），那么树看起来可能不一样。但是，只要元素集合相同，中序遍历以同样的顺序打印元素。中序遍历显示一个排好序的线性表。

✓ 复习题

25.1 显示将 44 插入图 25-4b 后的结果。

25.2 显示对图 25-1b 中二叉树中元素的中序、前序、后序遍历。

- 25.3 如果将由相同元素构成的集合以两个不同的次序插入 BST 中，这两棵对应的 BST 是否一样？中序遍历后是否一样？后序遍历后是否一样？前序遍历后是否一样？
- 25.4 在 BST 中插入一个元素的时间复杂度是多少？
- 25.5 使用递归实现 `search(element)` 方法。

25.3 删除 BST 中的一个元素

要点提示：为了从一棵二叉查找树中删除一个元素，首先需要定位该元素位置，然后在删除该元素以及重新连接树前，考虑两种情况——该结点有或者没有左子结点。

25.2.3 节给出了 `insert(element)` 方法。我们经常需要从二叉查找树中删除一个元素，这比向二叉查找树中添加一个元素复杂得多。

为了从一棵二叉查找树中删除一个元素，首先需要定位包含该元素的结点，以及它的父结点。假设 `current` 指向二叉查找树中包含该元素的结点，而 `parent` 指向 `current` 结点的父结点。`current` 结点可能是 `parent` 结点的左子结点，也可能是右子结点。这里需要考虑以下两种情况：

情况 1：当前结点没有左子结点，如图 25-10a 所示。这时只需要将该结点的父结点和该结点的右子结点相连，如图 25-10b 所示。

例如，为了在图 25-11a 中删除结点 10，需连接结点 10 的父结点和结点 10 的右子结点，如图 25-11b 所示。

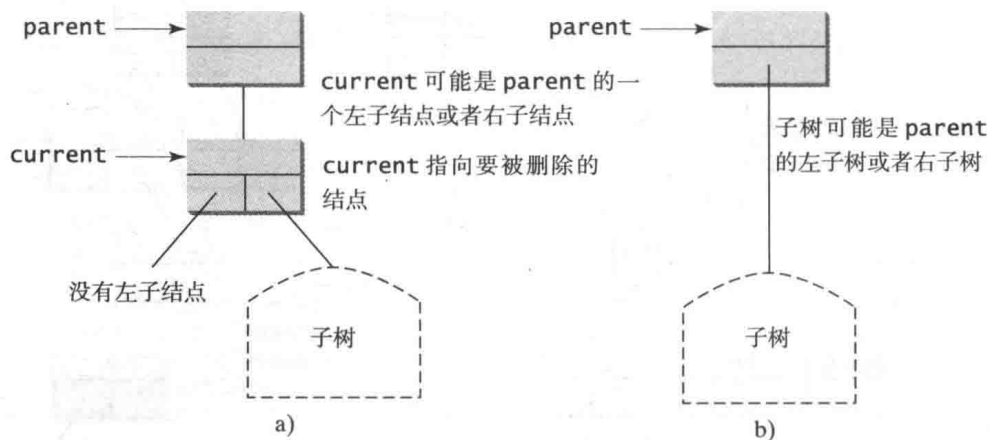


图 25-10 情况 1：当前结点没有左子结点

注意：如果当前结点是叶子结点，这就是属于情况 1。例如，为了删除图 25-11a 中的元素 16，将结点 16 的右孩子和它的父结点相连。在这种情况下，结点 16 的右孩子是 `null`。

情况 2：`current` 结点有左子结点。假设 `rightMost` 指向包含 `current` 结点的左子树中最大元素的结点，而 `parentOfRightMost` 指向 `rightMost` 结点的父结点，如图 25-12a 所示。注意，`rightMost` 结点不能有右子结点，但是可能会有左子结点。使用 `rightMost` 结点中的元素值替换 `current` 结点中的元素值，将 `parentOfRightMost` 结点和 `rightMost` 结点的左子结点相连，然后删除 `rightMost` 结点，如图 25-12b 所示。

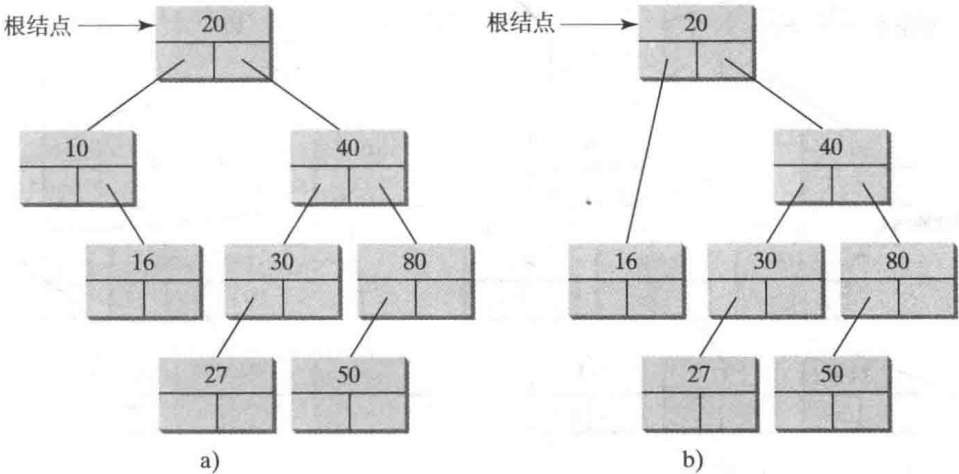


图 25-11 情况 1：从 a 中删除结点 10 得到 b

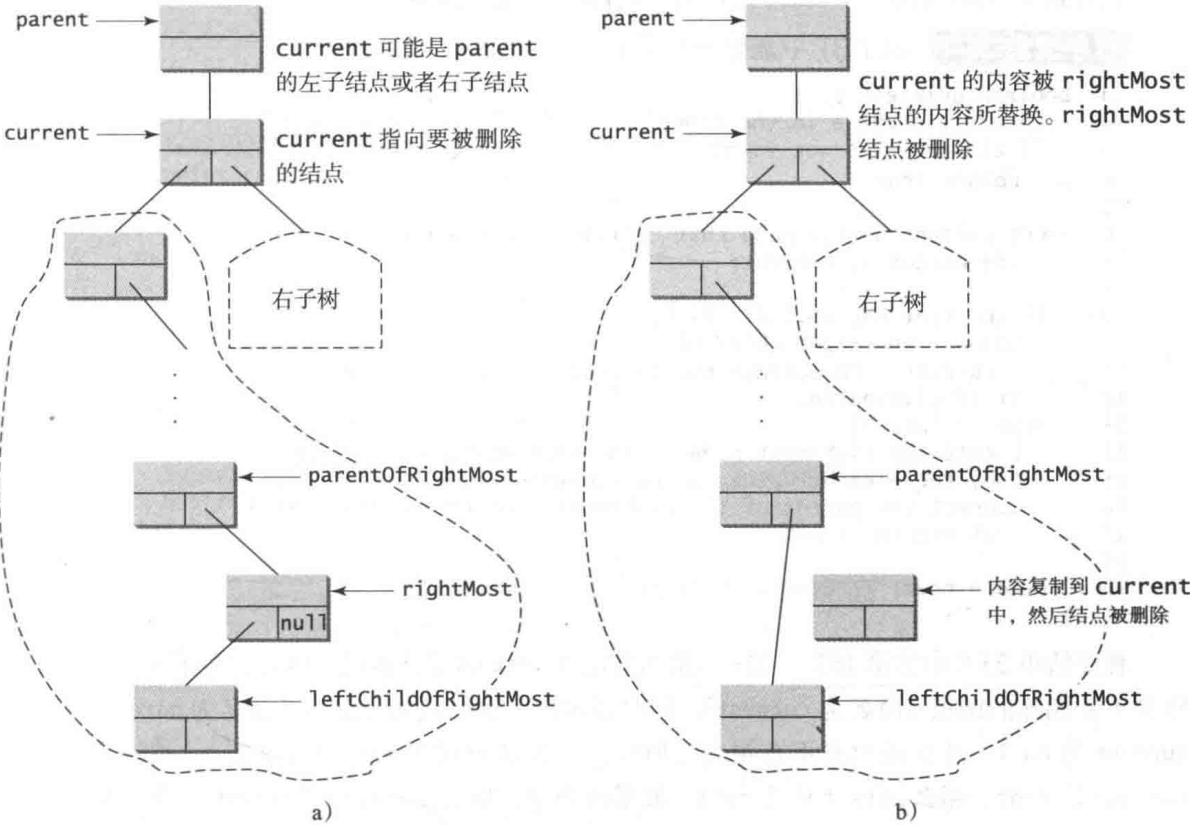


图 25-12 情况 2：当前结点有左子结点

例如，考虑删除图 25-13a 中的结点 20。rightMost 结点有一个值为 16 的元素。使用 current 结点中的 16 替换元素值 20，并将结点 10 作为结点 14 的父结点，如图 25-13b 所示。

注意：如果 current 的左子结点没有右子结点，那么 current.left 指向 current 左子树的大元素。在这种情况下，rightMost 是 current.left，而 parentOfRightMost 是 current。必须考虑这种特殊情况，重新连接 rightMost 的右子结点和 parentOf-RightMost。

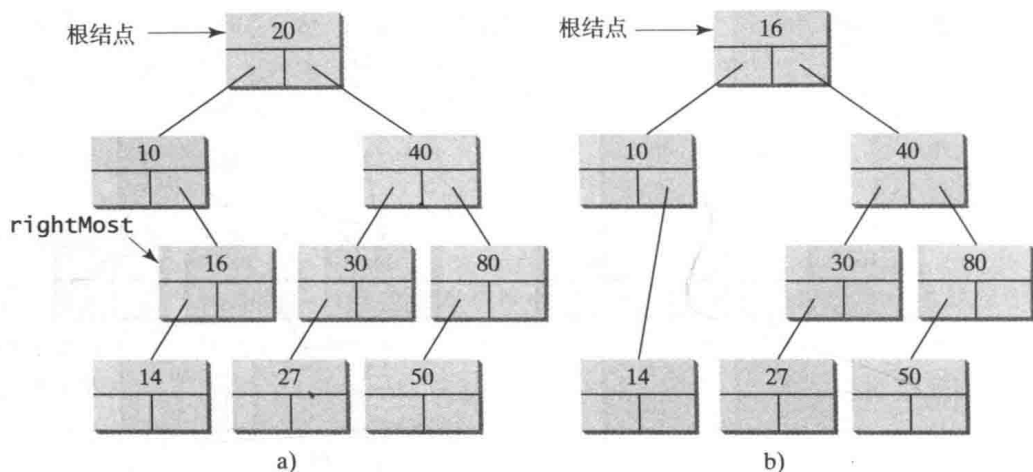


图 25-13 情况 2: 从 a 中删除结点 20 得到 b

程序清单 25-7 描述了从二叉查找树中删除一个元素的算法。

程序清单 25-7 从 BST 中删除一个元素

```

1  boolean delete(E e) {
2      Locate element e in the tree;
3      if element e is not found
4          return true;
5
6      Let current be the node that contains e and parent be
7          the parent of current;
8
9      if (current has no left child) // Case 1
10         Connect the right child of
11             current with parent; now current is not referenced, so
12             it is eliminated;
13     else // Case 2
14         Locate the rightmost node in the left subtree of current.
15         Copy the element value in the rightmost node to current.
16         Connect the parent of the rightmost node to the left child
17             of rightmost node;
18
19     return true; // Element deleted
20 }

```

程序清单 25-5 中的第 155 ~ 213 行给出方法 `delete` 的完整实现。该方法在第 157 ~ 170 行定位了要删除的结点 (命名为 `current`)，同时还定位了该结点的父结点 (命名为 `parent`)。如果 `current` 为 `null`，那么该元素不在树内。所以，该方法返回 `false` (第 173 行)。请注意，如果 `current` 是 `root`，那么 `parent` 就为 `null`。如果树为空，那么 `current` 和 `parent` 都为 `null`。

算法的情况 1 出现在第 176 ~ 187 行。在这种情况下，`current` 结点没有左子结点 (即 `current.left == null`)。如果 `parent` 为 `null`，就将 `current.right` 赋给 `root` (第 178 ~ 180 行)；否则，根据 `current` 是 `parent` 的左子结点还是右子结点，将 `current.right` 赋给 `parent.left` 或者 `parent.right` (第 182 ~ 185 行)。

算法的情况 2 出现在第 188 ~ 209 行。在这种情况下，`current` 结点有左子结点。算法定位当前结点的左子树最右端的结点 (命名为 `rightMost`)，并且定位它的父结点 (命名为 `parentOfRightMost`) (第 195 ~ 198 行)。用 `rightMost` 中的元素替换 `current` 中的元素 (第 201 行)。根据 `rightMost` 是 `parentOfRightMost` 的右子结点还是左子结点，将 `rightMost.left` 赋

给 `parentOfRightMost.right` 或者 `parentOfRightMost.left` (第 204 ~ 208 行)。

程序清单 25-8 给出从二叉查找树中删除一个元素的测试程序。

程序清单 25-8 TestBSTDelete.java

```
1 public class TestBSTDelete {
2     public static void main(String[] args) {
3         BST<String> tree = new BST<>();
4         tree.insert("George");
5         tree.insert("Michael");
6         tree.insert("Tom");
7         tree.insert("Adam");
8         tree.insert("Jones");
9         tree.insert("Peter");
10        tree.insert("Daniel");
11        printTree(tree);
12
13        System.out.println("\nAfter delete George:");
14        tree.delete("George");
15        printTree(tree);
16
17        System.out.println("\nAfter delete Adam:");
18        tree.delete("Adam");
19        printTree(tree);
20
21        System.out.println("\nAfter delete Michael:");
22        tree.delete("Michael");
23        printTree(tree);
24    }
25
26    public static void printTree(BST tree) {
27        // Traverse tree
28        System.out.print("Inorder (sorted): ");
29        tree.inorder();
30        System.out.print("\nPostorder: ");
31        tree.postorder();
32        System.out.print("\nPreorder: ");
33        tree.preorder();
34        System.out.print("\nThe number of nodes is " + tree.getSize());
35        System.out.println();
36    }
37 }
```

```
Inorder (sorted): Adam Daniel George Jones Michael Peter Tom
Postorder: Daniel Adam Jones Peter Tom Michael George
Preorder: George Adam Daniel Michael Jones Tom Peter
The number of nodes is 7
```

After delete George:

```
Inorder (sorted): Adam Daniel Jones Michael Peter Tom
Postorder: Adam Jones Peter Tom Michael Daniel
Preorder: Daniel Adam Michael Jones Tom Peter
The number of nodes is 6
```

After delete Adam:

```
Inorder (sorted): Daniel Jones Michael Peter Tom
Postorder: Jones Peter Tom Michael Daniel
Preorder: Daniel Michael Jones Tom Peter
The number of nodes is 5
```

After delete Michael:

```
Inorder (sorted): Daniel Jones Peter Tom
Postorder: Peter Tom Jones Daniel
Preorder: Daniel Jones Tom Peter
The number of nodes is 4
```


图 26-14 ~ 图 25-16 给出随着从树中删除元素树的演变过程。

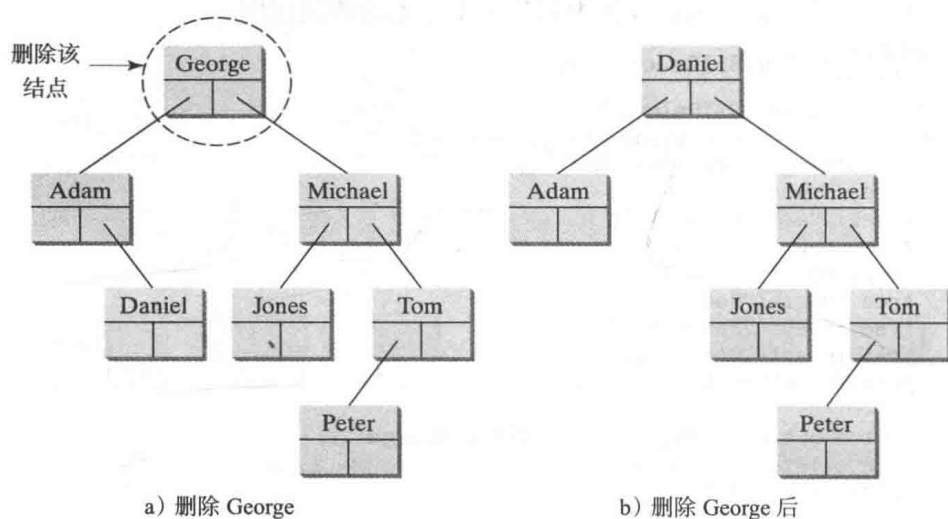


图 25-14 删除 George 属于情况 2

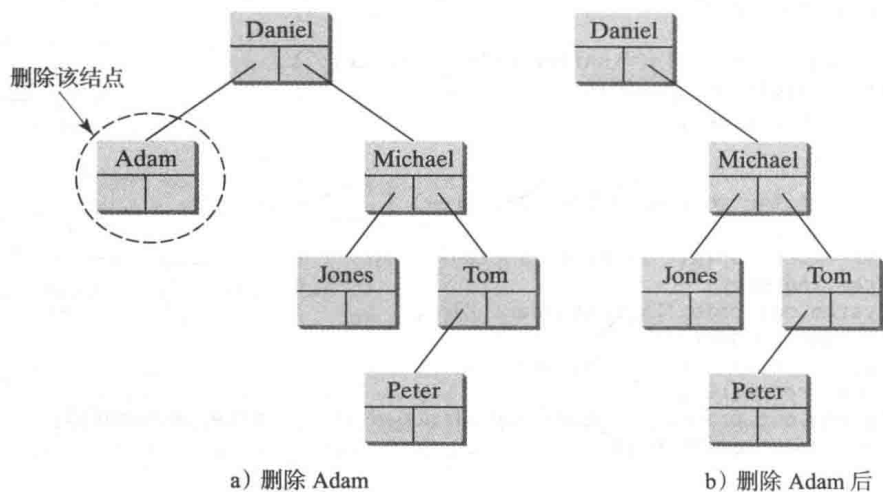


图 25-15 删除 Adam 属于情况 1

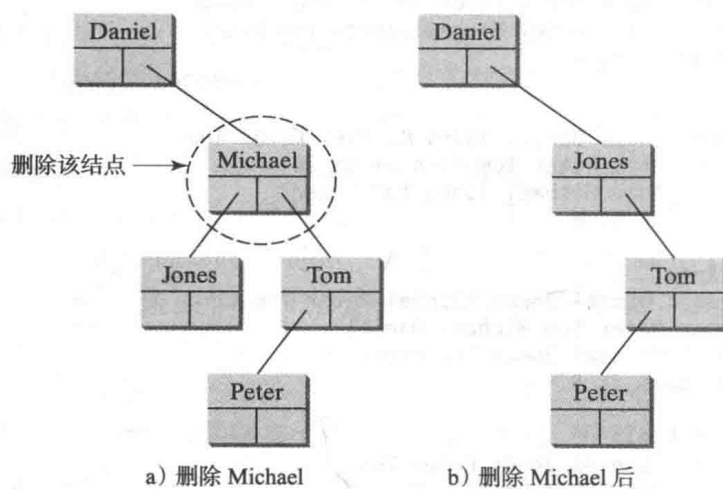


图 25-16 删除 Michael 属于情况 2

【1】注意：很明显中序遍历、前序遍历和后序遍历的时间复杂度都是 $O(n)$ ，因为每个结点只遍历一次。查找、插入和删除的时间复杂度是树的高度。在最差的情况下，树的高度为 $O(n)$ 。如果树是平衡的，高度将是 $O(\log n)$ 。我们将在第 26 章以及奖励章节第 40 和 41 章中介绍平衡二叉树。

复习题

- 25.6 显示从图 25-4b 所示的树中删除 55 之后的结果。
- 25.7 显示从图 25-4b 所示的树中删除 60 之后的结果。
- 25.8 从 BST 中删除一个元素的时间复杂度是多少？
- 25.9 如果将程序清单 25-5 中情况 2 第 204 ~ 208 行的 `delete()` 方法用下面的代码替换，算法还正确吗？

```
parentOfRightMost.right = rightMost.left;
```

25.4 树的可视化和 MVC

要点提示：可以应用递归来显示一棵二叉树。

【1】教学注意：数据结构课程面临的挑战是激发学生的兴趣。用图形显示二叉树不仅有助于学生理解二叉树的工作机制，而且还会激发学生对程序设计的兴趣。本节介绍可视化二叉树的技术。学生也可以在其他项目中应用可视化技术。

如何显示一棵二叉树？它是一种递归的结构，因此可以应用递归来显示一棵二叉树。可以简单显示根结点，然后递归地显示两棵子树。可以应用显示思瑞平斯基三角形（程序清单 18-9）的技术来显示二叉树。为了简单起见，我们假设键值是小于 100 的正整数。程序清单 25-9 和程序清单 25-10 给出该程序，而图 25-17 显示程序的一些运行示例。

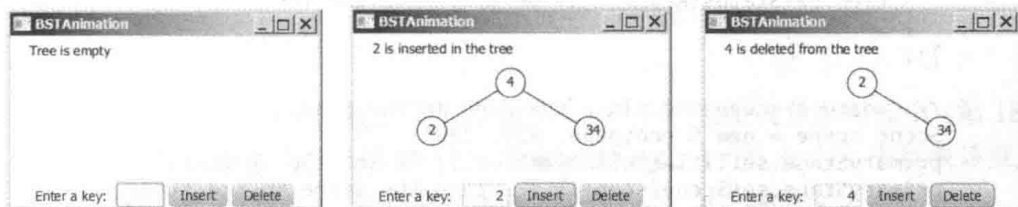


图 25-17 图形化显示一棵二叉树

程序清单 25-9 BSTAnimation.java

```
1 import javafx.application.Application;
2 import javafx.geometry.Pos;
3 import javafx.stage.Stage;
4 import javafx.scene.Scene;
5 import javafx.scene.control.Button;
6 import javafx.scene.control.Label;
7 import javafx.scene.control.TextField;
8 import javafx.scene.layout.BorderPane;
9 import javafx.scene.layout.HBox;
10
11 public class BSTAnimation extends Application {
12     @Override // Override the start method in the Application class
13     public void start(Stage primaryStage) {
14         BST<Integer> tree = new BST<>(); // Create a tree
15
16         BorderPane pane = new BorderPane();
```

```

17  BTreeView view = new BTreeView(tree); // Create a BTreeView
18  pane.setCenter(view);
19
20  TextField tfKey = new TextField();
21  tfKey.setPrefColumnCount(3);
22  tfKey.setAlignment(Pos.BASELINE_RIGHT);
23  Button btInsert = new Button("Insert");
24  Button btDelete = new Button("Delete");
25  HBox hBox = new HBox(5);
26  hBox.getChildren().addAll(new Label("Enter a key: "),
27      tfKey, btInsert, btDelete);
28  hBox.setAlignment(Pos.CENTER);
29  pane.setBottom(hBox);
30
31  btInsert.setOnAction(e -> {
32      int key = Integer.parseInt(tfKey.getText());
33      if (tree.search(key)) { // key is in the tree already
34          view.displayTree();
35          view.setStatus(key + " is already in the tree");
36      } else {
37          tree.insert(key); // Insert a new key
38          view.displayTree();
39          view.setStatus(key + " is inserted in the tree");
40      }
41  });
42
43  btDelete.setOnAction(e -> {
44      int key = Integer.parseInt(tfKey.getText());
45      if (!tree.search(key)) { // key is not in the tree
46          view.displayTree();
47          view.setStatus(key + " is not in the tree");
48      } else {
49          tree.delete(key); // Delete a key
50          view.displayTree();
51          view.setStatus(key + " is deleted from the tree");
52      }
53  });
54
55  // Create a scene and place the pane in the stage
56  Scene scene = new Scene(pane, 450, 250);
57  primaryStage.setTitle("BSTAnimation"); // Set the stage title
58  primaryStage.setScene(scene); // Place the scene in the stage
59  primaryStage.show(); // Display the stage
60  }
61  }

```

程序清单 25-10 BTreeView.java

```

1  import javafx.scene.layout.Pane;
2  import javafx.scene.paint.Color;
3  import javafx.scene.shape.Circle;
4  import javafx.scene.shape.Line;
5  import javafx.scene.text.Text;
6
7  public class BTreeView extends Pane {
8      private BST<Integer> tree = new BST<>();
9      private double radius = 15; // Tree node radius
10     private double vGap = 50; // Gap between two levels in a tree
11
12     BTreeView(BST<Integer> tree) {
13         this.tree = tree;
14         setStatus("Tree is empty");
15     }
16

```

```

17 public void setStatus(String msg) {
18     getChildren().add(new Text(20, 20, msg));
19 }
20
21 public void displayTree() {
22     this.getChildren().clear(); // Clear the pane
23     if (tree.getRoot() != null) {
24         // Display tree recursively
25         displayTree(tree.getRoot(), getWidth() / 2, vGap,
26             getWidth() / 4);
27     }
28 }
29
30 /** Display a subtree rooted at position (x, y) */
31 private void displayTree(BST.TreeNode<Integer> root,
32     double x, double y, double hGap) {
33     if (root.left != null) {
34         // Draw a line to the left node
35         getChildren().add(new Line(x - hGap, y + vGap, x, y));
36         // Draw the left subtree recursively
37         displayTree(root.left, x - hGap, y + vGap, hGap / 2);
38     }
39
40     if (root.right != null) {
41         // Draw a line to the right node
42         getChildren().add(new Line(x + hGap, y + vGap, x, y));
43         // Draw the right subtree recursively
44         displayTree(root.right, x + hGap, y + vGap, hGap / 2);
45     }
46
47     // Display a node
48     Circle circle = new Circle(x, y, radius);
49     circle.setFill(Color.WHITE);
50     circle.setStroke(Color.BLACK);
51     getChildren().addAll(circle,
52         new Text(x - 4, y + 4, root.element + ""));
53 }
54 }

```

程序清单 25-9 中，一棵树被创建（第 14 行），一个树视图放置在面板中（第 18 行）。在将一个新的键值插入树中之后（第 37 行），重新绘制这棵树（第 38 行）来反映该变化。在删除一个键值之后（第 49 行），重新绘制这棵树（第 50 行）来反映该变化。

程序清单 25-10 中，结点显示为一个半径 `radius` 为 15 的圆（第 48 行）。在树中，将两层之间的距离定义为 `vGap`，取值 50（第 25 行）。`hGap`（第 32 行）定义两个结点之间的水平距离。当递归调用 `displayTree` 方法时，该值在下一层中减半（`hGap/2`）（第 44 和 51 行）。注意，在树中没有改变 `vGap`。

如果子树不为空，那么 `displayTree` 方法被递归调用来显示一棵左子树（第 33 ~ 38 行）和一棵右子树（第 40 ~ 45 行）。一条直线添加到面板中来连接两个结点（第 35 和 42 行），注意方法先将直线添加到面板中，然后添加两个圆到面板中（第 52 行），这样圆会在直线之上绘制，从而获得较好的视觉效果。

程序假定键值都是整数。可以很容易修改程序，使得它针对泛型类型，可以显示字符或者短字符串的键值。

树的可视化是一个模型 - 视图 - 控制器（MVC）软件架构的例子。这是一个用于软件开发的重要架构。模型用于存储和处理数据，视图用于可视化地表达数据，控制器处理用户和模型的交互，并且控制视图，如图 25-18 所示。

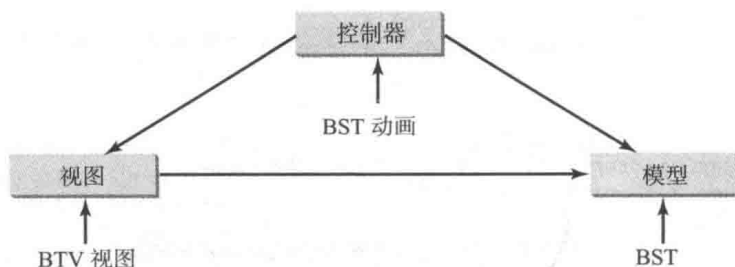


图 25-18 控制器获得数据并且将其存储在模型中。视图显示存储在模型中的数据

MVC 架构将数据的存储和处理与数据的可视化表达分离。它具有两个主要的好处：

- 使得多个视图成为可能，这样数据可以通过同样一个模型来分享。例如，你可以创建一个新的视图，将树显示为根结点在左边，而树水平向右生长（参见编程练习题 25.11）。
- 简化了编写复杂程序的任务，使得组件可扩展，并且易于维护。可以改变视图而不影响模型，反之亦然。

✓ 复习题

- 25.10 如果树为空，那么 `displayTree` 方法将被调用多少次？如果树有 100 个结点，那么 `displayTree` 方法将被调用多少次？
- 25.11 `displayTree` 方法以哪种顺序来访问树中的结点——中序、前序——还是后序？
- 25.12 如果 `BTreeView.java` 中第 47 ~ 52 行的代码移到第 33 行，将会发生什么情况？
- 25.13 什么是 MVC？MVC 的好处是什么？

25.5 迭代器

要点提示： BST 是可遍历的，因为它被定义为 `java.lang.Iterable` 接口的子类型。

方法 `inorder()`、`preorder()` 和 `postorder()` 分别以 `inorder`、`preorder` 和 `postorder` 方式显示二叉树中的元素。这些方法都局限于显示树中的元素。如果要处理二叉树中的元素，而不是显示它们，那么不能使用这些方法。回顾下遍历一个集合或者线性表的元素时提供了一个迭代器。可以以同样的方式将迭代器应用到一棵二叉树上，从而提供一种统一的方式来遍历二叉树中的元素。

`java.util.Iterator` 接口定义了 `iterator` 方法，该方法返回一个 `java.util.Iterator` 的实例。`java.util.Iterator` 接口（如图 25-19 所示）定义了迭代器的一般特性。

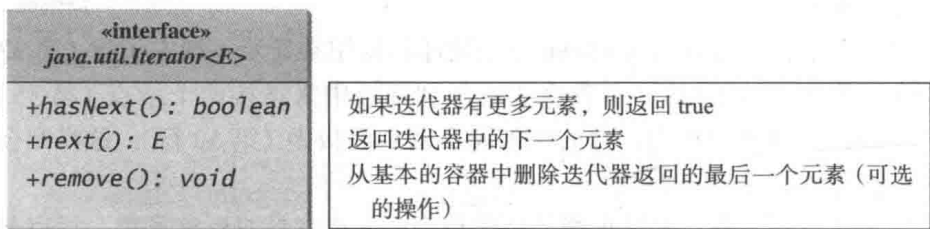


图 25-19 `Iterator` 接口定义遍历一个容器中元素的统一形式

`Tree` 接口继承自 `java.lang.Iterable`。由于 `BST` 是 `AbstractTree` 的子类，而 `AbstractTree` 实现了 `Tree`，所以 `BST` 是 `Iterable` 的子类型。`Iterable` 接口包含 `iterator()` 方法，

该方法返回 `java.util.Iterator` 的一个实例。

可以中序、前序或后序遍历二叉树。由于中序很常用，我们将使用中序来遍历一个二叉树中的元素。我们定义一个名为 `InorderIterator` 的迭代器类来实现 `java.util.Iterator` 接口，见程序清单 25-5（第 221 ~ 263 行）。`Iterator` 方法简单地返回一个 `InorderIterator` 的实例（第 217 行）。

`InorderIterator` 的构造方法调用 `inorder` 方法（第 228 行）。`inorder(root)` 方法（第 237 ~ 242 行）在 `list` 中存储树的所有元素。这些元素以 `inorder` 方式遍历。

一旦创建一个 `Iterator` 对象，它的 `current` 值将初始化为 0（第 225 行），它指向线性表中的第一个元素。调用 `next()` 方法返回当前元素，并将 `current` 移到指向线性表的下一个元素（第 253 行）。

方法 `hasNext()` 检查 `current` 是否在 `list` 的范围之内（第 246 行）。

方法 `remove()` 从树中删除当前元素（第 259 行）。在此之后，创建一个新的线性表（第 260 ~ 261 行）。注意，不需要改变 `current`。

程序清单 25-11 给出一个在 BST 中存储字符串的测试程序，并且显示所有字符串的大写形式。

程序清单 25-11 TestBSTWithIterator.java

```
1 public class TestBSTWithIterator {
2     public static void main(String[] args) {
3         BST<String> tree = new BST<>();
4         tree.insert("George");
5         tree.insert("Michael");
6         tree.insert("Tom");
7         tree.insert("Adam");
8         tree.insert("Jones");
9         tree.insert("Peter");
10        tree.insert("Daniel");
11
12        for (String s: tree)
13            System.out.print(s.toUpperCase() + " ");
14    }
15 }
```

ADAM DANIEL GEORGE JONES MICHAEL PETER TOM
--

`foreach` 循环（第 12 ~ 13 行）使用了一个迭代器来遍历树中的所有元素。

设计指南：迭代器是一个重要的软件设计模式。它提供遍历容器内元素的统一方法，同时隐藏该容器的构造细节。通过实现相同的接口 `java.util.Iterator`，可以编写一个程序，以相同的方式遍历所有容器的元素。

注意：`java.util.Iterator` 定义一个前向迭代器，它以前向的方向遍历迭代器中的元素，每个元素只能遍历一次。Java API 还提供 `java.util.ListIterator`，它支持前向遍历和后向遍历。如果你的数据结构要保证遍历的灵活性，可以将迭代器类定义为 `java.util.ListIterator` 的一个子类。

迭代器的实现不是很高效。每次通过迭代器删除一个元素时，整个线性表都要重新构造（程序清单 25-5 中第 261 行）。客户程序应该总是采用 BST 类中的 `delete` 方法来删除一个元素。为了防止用户使用迭代器中的 `remove` 方法，如下实现迭代器：

```
public void remove() {
    throw new UnsupportedOperationException
```

```
    ("Removing an element from the iterator is not supported");  
}
```

在使得 `remove` 方法不被迭代器类支持后，无须为树中的元素维护一个线性表使得迭代器更加高效。可以使用栈来存储结点，栈顶端的结点包含从 `next()` 方法返回的元素。如果树是平衡的，最大的栈的大小将为 $O(\log n)$ 。

✓ 复习题

- 25.14 什么是迭代器？
- 25.15 `java.lang.Iterable<E>` 接口中定义了什么方法？
- 25.16 假设你从程序清单 25-3 的第 1 行删除了 `extends Iterable<E>`，程序清单 25-11 还能编译吗？
- 25.17 定义为 `Iterable<E>` 的子类型的好处是什么？

25.6 示例学习：数据压缩

🔑 要点提示：霍夫曼编码通过使用更少的比特对经常出现的字符编码来压缩数据。字符的编码是基于字符在文本中出现的次数使用二叉树来构建的，该树称为霍夫曼编码树。

压缩数据是一个常见的任务。压缩文件的应用很多，本节介绍 David Huffman 在 1952 年发明的霍夫曼编码。

在 ASCII 码中，每个字符都被编码为 8 比特。如果一个文本中包含 100 个字符，则需要 800 比特来表示该文本。霍夫曼编码通过使用较少的比特对文本中常用的字符编码，以及更多的比特来对不常用的字符编码来减少文件的整个大小。霍夫曼编码中，字符的编码是基于字符在文本中出现的次数使用二叉树来构建的，该树称为霍夫曼编码树（Huffman coding tree）。假设该文本是 `Mississippi`，它的霍夫曼树就如图 25-20a 所示。结点的左边和右边分别被赋值 0 和 1。每个字符都是树中的一个叶结点。字符的编码由从根到叶结点的路径上的边的值所组成，如图 25-20b 所示。因为文本中 `i` 和 `s` 出现得比 `M` 和 `p` 多，所以它们都被赋予更短的代码。

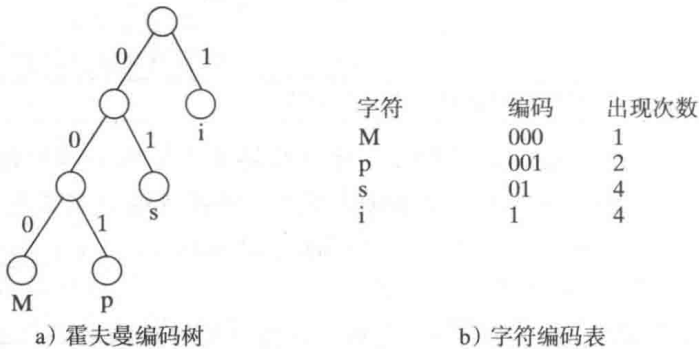


图 25-20 使用编码树基于字符在文本中出现的次数来构建字符的编码

基于图 25-20 所示的编码方案，

编码为
Mississippi =====> 000101011010110010011 =====> 解码为
Mississippi

编码树也用于将一个比特序列解码为一个文本。为了做到这点，从序列中的第一个比特开始，基于该比特值决定是走向树的根结点的左分支还是右分支。考虑下一个比特，然后继续基于该比特值决定是走向左分支还是右分支。当到达一个叶子结点时，就找到了一个字

符。数据流中的下一个比特就是下一个字符的第一个比特。例如，数据流 011001 被解码为 sip，其中 01 匹配 s，1 匹配 i，001 匹配 p。

为了构建一棵霍夫曼编码树，使用如下算法：

1) 从由树构成的森林开始。每棵树都包含一个字符结点。每个结点的权重就是文本中字符的出现次数。

2) 重复以下步骤来合并树，直到只有一棵树为止：选择两棵有最小权重的树，创建一个新结点作为它们的父结点。这棵新树的权重是子树的权重和。

3) 对于每个内部结点，给它的左边赋值 0，而给它的右边赋值 1。所有的叶子结点都表示文本中的字符。

下面是一个为文本 Mississippi 构建编码树的例子。字符的出现次数表如图 25-20b 所示。初始情况下，森林包含单结点树，如图 25-21a 所示。重复组合树以形成大树，直到只留下一棵树，如图 25-21b ~ 图 25-21d 所示。

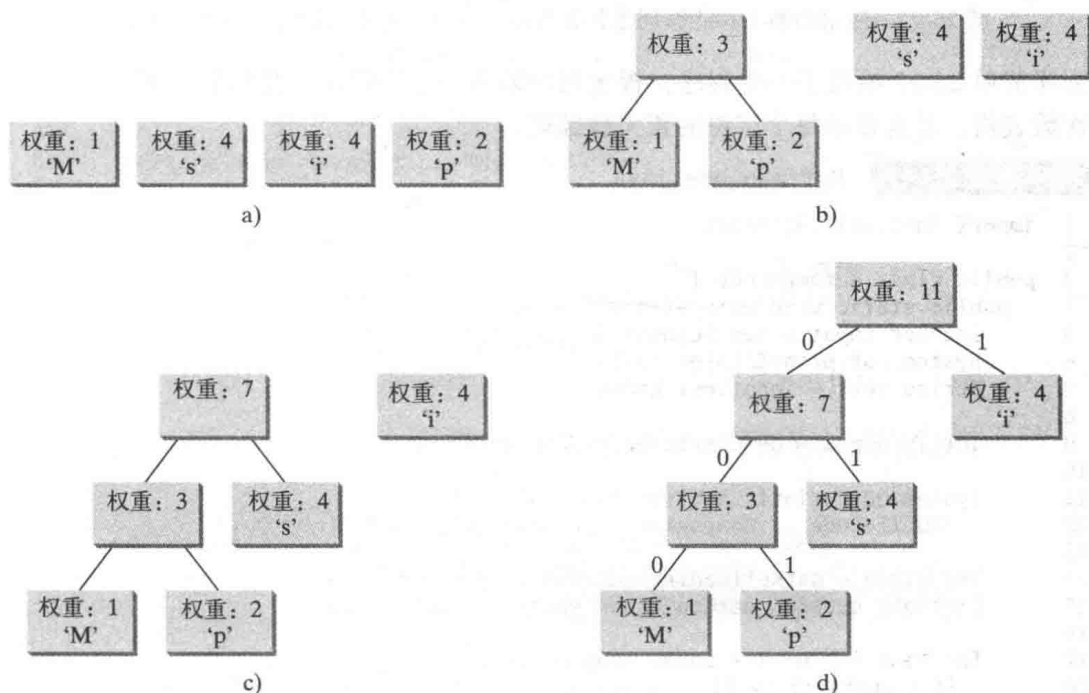


图 25-21 通过重复地组合两棵最小权重的树来构建编码树

值得注意的是，没有编码是另外一个编码的前缀。整个属性保证了流可以无二义性地解码。

教学注意： 参见链接 www.cs.armstrong.edu/liang/animation/HuffmanCodingAnimation.html 来查看霍夫曼编码是如何工作的交互式 GUI 演示，如图 25-22 所示。

这里使用的算法是贪婪算法 (greedy algorithm) 的一个示例。贪婪算法经常用于解决优化问题。算法做出局部最优的选择，并希望这样的选择会导致全局最优。这个示例中，算法总是选择具有最小权重的两棵树，并且创建一个新的结点作为它们的父结点。这种凭直觉的最优局部解的确引向了最后构造霍夫曼树的最优解。作为另外一个示例，考虑将钱兑换为可能的最少硬币。一种贪婪算法将优先使用最大的可能硬币。例如，对于 98 美分，将使用三个 quarter (25 美分) 来凑成 75 美分，然后加上两个 dime (10 美分) 来凑成 95 美分，再加

上 3 个 penny (美分) 来凑成 98 美分。贪婪算法找到了该问题的一个最优解。然而, 贪婪算法并不是总能找到最优的结果。参见编程练习题 25.22 的装箱问题。

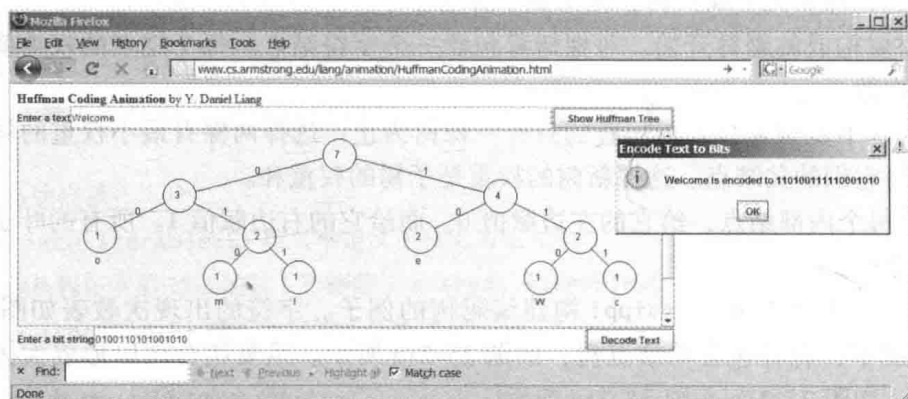


图 25-22 使用动画工具能够创建并查看霍夫曼树, 并用该树完成编码和解码

程序清单 25-12 给出了一个程序, 提示用户输入一个字符串, 然后显示文本中的字符的出现次数表格, 并且显示每个字符的霍夫曼编码。

程序清单 25-12 HuffmanCode.java

```

1  import java.util.Scanner;
2
3  public class HuffmanCode {
4      public static void main(String[] args) {
5          Scanner input = new Scanner(System.in);
6          System.out.print("Enter text: ");
7          String text = input.nextLine();
8
9          int[] counts = getCharacterFrequency(text); // Count frequency
10
11          System.out.printf("%-15s%-15s%-15s%-15s\n",
12                          "ASCII Code", "Character", "Frequency", "Code");
13
14          Tree tree = getHuffmanTree(counts); // Create a Huffman tree
15          String[] codes = getCode(tree.root); // Get codes
16
17          for (int i = 0; i < codes.length; i++)
18              if (counts[i] != 0) // (char)i is not in text if counts[i] is 0
19                  System.out.printf("%-15d%-15s%-15d%-15s\n",
20                                  i, (char)i + "", counts[i], codes[i]);
21      }
22
23      /** Get Huffman codes for the characters
24       * This method is called once after a Huffman tree is built
25       */
26      public static String[] getCode(Tree.Node root) {
27          if (root == null) return null;
28          String[] codes = new String[2 * 128];
29          assignCode(root, codes);
30          return codes;
31      }
32
33      /** Recursively get codes to the leaf node */
34      private static void assignCode(Tree.Node root, String[] codes) {
35          if (root.left != null) {
36              root.left.code = root.code + "0";

```

```

37     assignCode(root.left, codes);
38
39     root.right.code = root.code + "1";
40     assignCode(root.right, codes);
41 }
42 else {
43     codes[(int)root.element] = root.code;
44 }
45 }
46
47 /** Get a Huffman tree from the codes */
48 public static Tree getHuffmanTree(int[] counts) {
49     // Create a heap to hold trees
50     Heap<Tree> heap = new Heap<>(); // Defined in Listing 23.9
51     for (int i = 0; i < counts.length; i++) {
52         if (counts[i] > 0)
53             heap.add(new Tree(counts[i], (char)i)); // A leaf node tree
54     }
55
56     while (heap.getSize() > 1) {
57         Tree t1 = heap.remove(); // Remove the smallest-weight tree
58         Tree t2 = heap.remove(); // Remove the next smallest
59         heap.add(new Tree(t1, t2)); // Combine two trees
60     }
61
62     return heap.remove(); // The final tree
63 }
64
65 /** Get the frequency of the characters */
66 public static int[] getCharacterFrequency(String text) {
67     int[] counts = new int[256]; // 256 ASCII characters
68
69     for (int i = 0; i < text.length(); i++)
70         counts[(int)text.charAt(i)]++; // Count the characters in text
71
72     return counts;
73 }
74
75 /** Define a Huffman coding tree */
76 public static class Tree implements Comparable<Tree> {
77     Node root; // The root of the tree
78
79     /** Create a tree with two subtrees */
80     public Tree(Tree t1, Tree t2) {
81         root = new Node();
82         root.left = t1.root;
83         root.right = t2.root;
84         root.weight = t1.root.weight + t2.root.weight;
85     }
86
87     /** Create a tree containing a leaf node */
88     public Tree(int weight, char element) {
89         root = new Node(weight, element);
90     }
91
92     @Override /** Compare trees based on their weights */
93     public int compareTo(Tree t) {
94         if (root.weight < t.root.weight) // Purposely reverse the order
95             return 1;
96         else if (root.weight == t.root.weight)
97             return 0;
98         else
99             return -1;
100     }

```

```

101
102     public class Node {
103         char element; // Stores the character for a leaf node
104         int weight; // weight of the subtree rooted at this node
105         Node left; // Reference to the left subtree
106         Node right; // Reference to the right subtree
107         String code = ""; // The code of this node from the root
108
109         /** Create an empty node */
110         public Node() {
111         }
112
113         /** Create a node with the specified weight and character */
114         public Node(int weight, char element) {
115             this.weight = weight;
116             this.element = element;
117         }
118     }
119 }
120 }

```

Enter text: Welcome <input type="button" value="Enter"/>			
ASCII Code	Character	Frequency	Code
87	W	1	110
99	c	1	111
101	e	2	10
108	l	1	011
109	m	1	010
111	o	1	00

该程序提示用户输入一个文本字符串（第 5 ~ 7 行），然后计算文本中字符的出现次数（第 9 行）。getCharacterFrequency 方法（第 66 ~ 73 行）创建一个数组 counts 来统计文本中 256 个 ASCII 字符每一个的出现次数。如果文本中出现一个字符，它对应的计数器就加 1（第 70 行）。

程序基于 counts 获取霍夫曼编码树（第 14 行）。这棵树由链式结点构成。Node 类定义在第 102 ~ 118 行。每个结点都包含属性 element（存储字符）、weight（存储该结点下的子树的权重）、left（到左子树的链接）、right（到右子树的链接）和 code（存储该字符的霍夫曼编码）。Tree 类（第 76 ~ 119 行）包含根结点的属性。可以从该根结点访问树中的所有结点。Tree 类实现了 Comparable。这些树是基于它们的权重来进行比较的。比较顺序被故意颠倒（第 93 ~ 100 行），因此，最小权重的树首先从树的堆中删除。

方法 getHuffmanTree 返回一棵霍夫曼编码树。初始情况下，创建单结点树并将其添加到堆中（第 50 ~ 54 行）。在 while 循环的每次迭代中（第 56 ~ 60 行），将两棵最小权重的树从堆中删除，然后将它们组合成一棵大树，接着将新树添加到堆中。这个过程持续到堆中只包含一棵树为止，这就是我们给出的文本最终的霍夫曼树。

方法 assignCode 给树中的每个结点赋予编码（第 34 ~ 45 行）。方法 getCode 获取每个叶子结点中字符的编码（第 26 ~ 31 行）。元素 codes[i] 包含字符 (char)i 的编码，其中 i 从 0 到 255。注意，如果 (char)i 不在文本中，那么 codes[i] 为 null。

复习题

- 25.18 霍夫曼树中的每个内部结点具有两个子结点，对吗？
- 25.19 什么是贪婪算法？举一个例子。
- 25.20 如果程序清单 25-10 中第 50 行的 Heap 类替换为 java.util.PriorityQueue，程序还能工作吗？

关键术语

binary search tree (二叉查找树)

binary tree (二叉树)

breadth-first traversal (广度优先遍历)

depth-first traversal (深度优先遍历)

greedy algorithm (贪婪算法)

Huffman coding (霍夫曼编码)

inorder traversal (中序遍历)

postorder traversal (后序遍历)

preorder traversal (前序遍历)

tree traversal (树的遍历)

本章小结

1. 二叉查找树 (BST) 是一种分层的数据结构。学习了如何定义和实现 BST 类。学习了如何向 / 从 BST 插入和删除元素。学习了如何使用中序、后序、前序、深度优先以及广度优先搜索来遍历 BST。
2. 迭代器是一个提供了遍历像集合、线性表或二叉树这样的容器中的元素的统一方法的对象。学习了如何定义和实现遍历二叉树中元素的迭代器类。
3. 霍夫曼编码是一种压缩数据的方案，它使用较少的比特来编码经常出现的字符。字符的编码是使用二叉树基于它在文本中出现的次数来构建的，该二叉树称为霍夫曼编码树。

测试题

回答位于网址 www.cs.armstrong.edu/liang/intro10e/quiz.html 的本章测试题。

编程练习题

25.2 ~ 25.6 节

- *25.1 (在 BST 中添加新方法) 向 BST 类中添加以下新方法:

```
/** Displays the nodes in a breadth-first traversal */
public void breadthFirstTraversal()
```

```
/** Returns the height of this binary tree */
public int height()
```

- *25.2 (测试完全二叉树) 完全二叉树是指叶子结点都在同一层的二叉树。在 BST 类中添加一个方法，如果这棵树是完全二叉树，返回 true。

(提示: 完全二叉树中的结点个数是 $2^{\text{depth}-1}$ 。)

```
/** Returns true if the tree is a full binary tree */
boolean isFullBST()
```

- **25.3 (不使用递归实现中序遍历) 使用栈替代递归，实现 BST 中的 inorder 方法。编写一个测试程序，提示用户输入 10 个整数，将它们保存在一个 BST 中，调用 inorder 方法来显示这些元素。

- **25.4 (不使用递归实现前序遍历) 使用栈替代递归，实现 BST 中的 preorder 方法。编写一个测试程序，提示用户输入 10 个整数，将它们保存在一个 BST 中，调用 preorder 方法来显示这些元素。

- **25.5 (不使用递归实现后序遍历) 使用栈替代递归，实现 BST 中的 postorder 方法。编写一个测试程序，提示用户输入 10 个整数，将它们保存在一个 BST 中，调用 postorder 方法来显示这些元素。

- **25.6 (找出叶子结点) 在 BST 类中添加一个方法，返回叶子结点的个数，如下所示:

```
/** Returns the number of leaf nodes */
public int getNumberOfLeaves()
```

****25.7** (找出非叶子结点) 在 BST 类中添加一个方法, 返回非叶子结点的个数, 如下所示:

```
/** Returns the number of nonleaf nodes */
public int getNumberOfNonLeaves()
```

*****25.8** (实现双向迭代器) java.util.Iterator 接口定义了一个前向迭代器。Java API 也提供定义了一个定义双向迭代器的 java.util.ListIterator 接口。研究 ListIterator 并定义一个 BST 类的双向迭代器。

****25.9** (树的 clone 和 equals 方法) 实现 BST 类中的 clone 和 equals 方法。两棵 BST 树如果包含相同的元素, 则它们是相等的。clone 方法返回一棵 BST 树的完全一样的一个副本。

25.10 (前序迭代器) 添加以下方法到 BST 类中, 返回一个迭代器, 用于前序遍历 BST 中的元素。

```
/** Returns an iterator for traversing the elements in preorder */
java.util.Iterator<E> preorderIterator()
```

25.11 (显示树) 编写一个新的视图类, 水平显示树, 根在左边, 如图 25-23 所示。

****25.12** (测试 BST) 设计和编写一个完整的测试程序, 测试程序清单 25-5 中的 BST 类是否符合所有要求。

****25.13** (在 BSTAnimation 中添加新按钮) 修改程序清单 25-9, 添加三个新按钮——Show Inorder、Show Preorder 和 Show Postorder——以便在标签中显示结果, 如图 25-24 所示。还需要修改 BST.java 来实现 inorderList ()、preorderList () 和 postorderList () 方法, 这样, 这些方法就能以中序、前序和后序返回一个由结点元素构成的 List, 如下所示:

```
public java.util.List<E> inorderList();
public java.util.List<E> preorderList();
public java.util.List<E> postorderList();
```

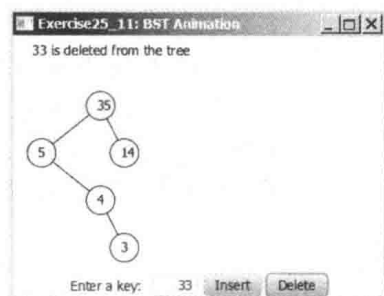


图 25-23 一棵二叉树水平显示

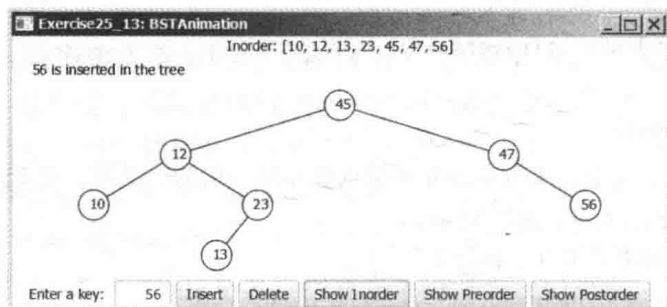


图 25-24 当单击图中的 Show Inorder、Show Preorder 或者 Show Postorder 按钮, 就会在标签中分别以中序、前序和后序显示元素

***25.14** (使用 Comparator 的泛型 BST) 修改程序清单 25-5 中的 BST, 使用泛型参数和一个 Comparator 来比较对象。定义一个构造方法, 使用 Comparator 作为它的参数, 如下所示:

```
BST(Comparator<? super E> comparator)
```

*****25.15** (BST 的父引用) 通过添加一个到某结点的父结点的引用来重新定义 TreeNode, 如下所示:

```
BST.TreeNode<E>
#element: E
#left: TreeNode<E>
#right: TreeNode<E>
#parent: TreeNode<E>
```

重新实现 BST 类中的 insert 和 delete 方法，为树中的每个结点更新父结点。在 BST 中添加以下新方法：

```
/** Returns the node for the specified element.
 * Returns null if the element is not in the tree. */
private TreeNode<E> getNode(E element)

/** Returns true if the node for the element is a leaf */
private boolean isLeaf(E element)

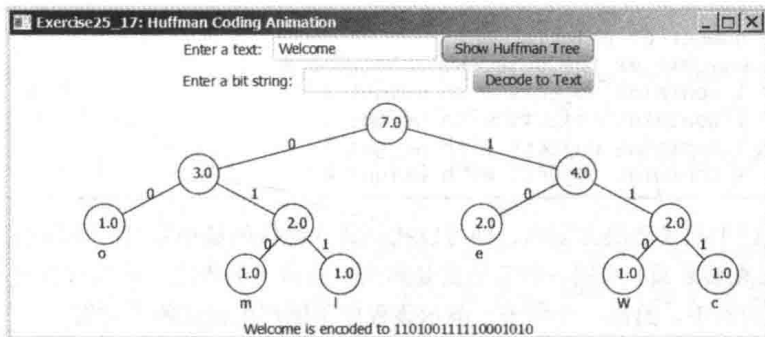
/** Returns the path of elements from the specified element
 * to the root in an array list. */
public ArrayList<E> getPath(E e)
```

编写一个测试程序，提示用户输入 10 个整数，将它们添加到树中，从树中删除第一个整数，然后显示到所有叶子结点的路径。下面是一个运行示例：

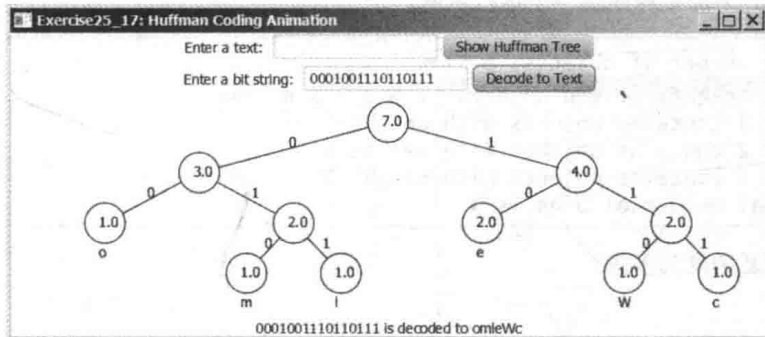
```
Enter 10 integers: 45 54 67 56 50 45 23 59 23 67  Enter
[50, 54, 23]
[59, 56, 67, 54, 23]
```

***25.16 (数据压缩：霍夫曼编码) 编写一个程序，提示用户输入一个文件名，显示文件中字符出现次数的表格，然后显示每个字符的霍夫曼编码。

***25.17 (数据压缩：霍夫曼编码的动画) 编写一个程序，允许用户输入一个文本，然后显示基于该文本的霍夫曼编码树，如图 25-25a 所示。显示在一棵子树根结点的环中的子树的权重，显示每个叶子结点的字符，在标签中显示文本被编码后的比特。当用户单击 Decode Text 按钮时，一个比特字符串被解码为一个文本，显示在标签中，如图 25-25b 所示。



a)



b)

图 25-25 a) 动画中显示给定文本的编码树，文本编码的比特显示在标签中；b) 输入一个比特串，在标签中显示对应的文本

***25.18 (压缩一个文件) 编写一个程序，使用霍夫曼编码将源文件压缩为目标文件。首先使用

ObjectOutputStream 将霍夫曼编码输出到目标文件中，然后使用编程练习题 17.17 的 BitOutputStream 输出编码后的二进制内容到目标文件中。通过命令行传递文件信息：

```
java Exercise25_18 sourcefile targetfile
```

***25.19 (解压缩一个文件) 前一个练习题压缩一个文件。压缩的文件包含了霍夫曼编码以及压缩的内容。编写一个程序，使用以下命令将一个源文件解压缩为目标文件：

```
java Exercise25_19 sourcefile targetfile
```

25.20 (应用首次满足法解决装箱问题) 编写一个程序，将各种重量的物体装箱到容器中。每个容器可以容纳最多 10 磅。程序使用贪婪算法，将物体放置在它可以放下的第一个箱中。程序应该提示用户输入物体的总数以及每个物体的重量。程序显示需要装入物体的容器总数以及每个容器的内容。下面是一个程序的运行示例：

```
Enter the number of objects: 6
Enter the weights of the objects: 7 5 2 3 5 8
Container 1 contains objects with weight 7 2
Container 2 contains objects with weight 5 3
Container 3 contains objects with weight 5
Container 4 contains objects with weight 8
```

该程序可以产生最优解决方案吗，即可以找到装入物体的最小数目的容器吗？

25.21 (最小物体优先的装箱) 采用一种新的贪婪算法重写前面的程序，将具有最小重量的物体放置在它首先适应的箱中。程序应该提示用户输入物体的总数以及每个物体的重量。程序显示需要装入物体的容器总数以及每个容器的内容。下面是一个程序的运行示例：

```
Enter the number of objects: 6
Enter the weights of the objects: 7 5 2 3 5 8
Container 1 contains objects with weight 2 3 5
Container 2 contains objects with weight 5
Container 3 contains objects with weight 7
Container 4 contains objects with weight 8
```

该程序可以产生最优解决方案吗，即可以找到装入物体的最小数目的容器吗？

25.22 (最大物体优先的装箱) 采用一种新的贪婪算法重写前面的程序，将具有最大重量的物体放置在它首先适应的箱中。给出一个例子，演示该程序不能产生最优解决方案。

25.23 (最优装箱) 重写前面的程序，使得它可以找到最优解决方案，可以使用最小数目的容器来装箱所有的物体。下面是程序的一个运行示例：

```
Enter the number of objects: 6  Enter
Enter the weights of the objects: 7 5 2 3 5 8  Enter
Container 1 contains objects with weight 7 3
Container 2 contains objects with weight 5 5
Container 3 contains objects with weight 2 8
The optimal number of bins is 3
```

程序的时间复杂度为多少？

AVL 树

【】 教学目标

- 了解什么是 AVL 树 (26.1 节)。
- 理解如何使用 LL 旋转、LR 旋转、RR 旋转以及 RL 旋转来重新平衡一棵树 (26.2 节)。
- 继承 BST 类, 设计 AVLTree (26.3 节)。
- 在 AVL 树中插入元素 (26.4 节)。
- 实现树的重新平衡 (26.5 节)。
- 从 AVL 树中删除元素 (26.6 节)。
- 实现 AVLTree 类 (26.7 节)。
- 测试 AVLTree 类 (26.8 节)。
- 分析在 AVL 树中查找、插入和删除操作的复杂度 (26.9 节)。

26.1 引言

🔑 要点提示: AVL 树是平衡二叉查找树。

第 25 章介绍了二叉查找树。二叉树的查找、插入和删除操作的时间依赖于树的高度。最坏情形下, 高度为 $O(n)$ 。如果一棵树是完全平衡的 (perfectly balanced) ——即一棵完全二叉树——它的高度是 $\log n$ 。可以保持一棵完全平衡的树吗? 可以的, 但是这样做的代价比较大。一个妥协的做法是保持一棵良好平衡的树——即每个结点的两个子树的高度基本一样。本章介绍 AVL 树。Web 章节 40 和 41 介绍 2-4 树和红黑树。

AVL 树是良好平衡的。AVL 树由两个俄罗斯计算机学家 G. M. Adelson-Velsky 和 E. M. Landis (因此命名为 AVL) 于 1962 年发明。在一棵 AVL 树中, 每个结点的子树的高度差距为 0 或者 1。可以得出一个 AVL 树的最大高度为 $O(\log n)$ 。

在一棵 AVL 树中插入或者删除一个元素的过程与在一棵普通二叉查找树中一样, 不同的是必须在插入或者删除操作之后进行重新平衡。一个结点的平衡因子 (balance factor) 是它右子树的高度减去左子树的高度。如果一个结点的平衡因子为 -1、0 或者 1, 那么这个结点是平衡的 (balanced)。如果结点的平衡因子为 -1, 则该结点被认为是左偏重 (left-heavy) 的, 如果平衡因子为 +1, 则认为是右偏重 (right-heavy) 的。

【】 教学注意: 可以参见网址 www.cs.armstrong.edu/liang/animation/web/AVLTree.html 看到 AVL 树如何工作的交互式的 GUI 演示, 如图 26-1 所示。

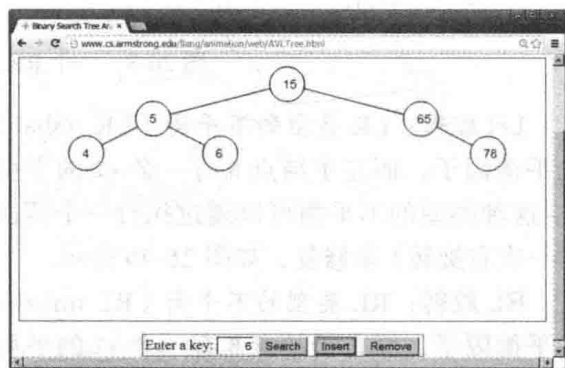


图 26-1 可以插入、删除和查找元素的动画工具

26.2 重新平衡树

要点提示：从 AVL 树中插入或者删除一个元素后，如果树变得不平衡了，执行一个旋转操作来重新平衡该树。

如果一个结点在插入或者删除操作后不平衡了，需要重新进行平衡。一个结点的重新获得平衡的过程称为旋转 (rotation)。有 4 种可能的旋转：LL、RR、LR 以及 RL。

LL 旋转：LL 类型的不平衡 (LL imbalance) 发生在结点 A 的如下情况上，A 有一个 -2 的平衡因子，而左子结点 B 有一个 -1 或者 0 的平衡因子，如图 26-2a 所示。这种类型的不平衡可以通过执行一个 A 上的简单右旋转来修复，如图 26-2b 所示。

RR 旋转：RR 类型的不平衡 (RR imbalance) 发生在结点 A 的如下情况上，A 有一个 +2 的平衡因子，而右子结点 B 有一个 +1 或者 0 的平衡因子，如图 26-3a 所示。这种类型的不平衡可以通过执行一个 A 上的简单左旋转来修复，如图 26-3b 所示。

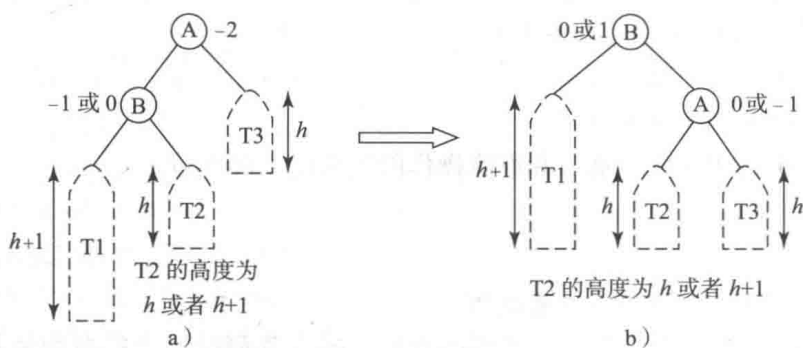


图 26-2 一个 LL 旋转修复 LL 不平衡

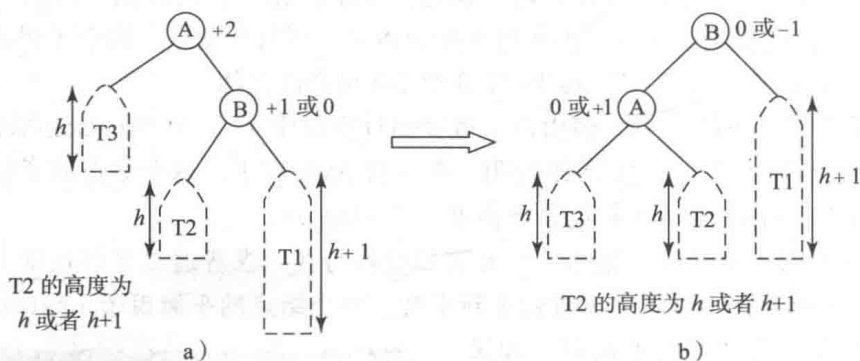


图 26-3 一个 RR 旋转修复 RR 不平衡

LR 旋转：LR 类型的不平衡 (LR imbalance) 发生在结点 A 的如下情况上，A 有一个 -2 的平衡因子，而左子结点 B 有一个 +1 的平衡因子，如图 26-4a 所示。假设 B 的右子结点为 C。这种类型的不平衡可以通过执行一个两次旋转（首先在 B 上的一次左旋转，然后在 A 上的一次右旋转）来修复，如图 26-4b 所示。

RL 旋转：RL 类型的不平衡 (RL imbalance) 发生在结点 A 的如下情况上，A 有一个 +2 的平衡因子，而右子结点 B 有一个 -1 的平衡因子，如图 26-5a 所示。假设 B 的左子结点为 C。这种类型的不平衡可以通过执行一个两次旋转（首先在 B 上的一次右旋转，然后在 A 上的一次左旋转）来修复，如图 26-5b 所示。

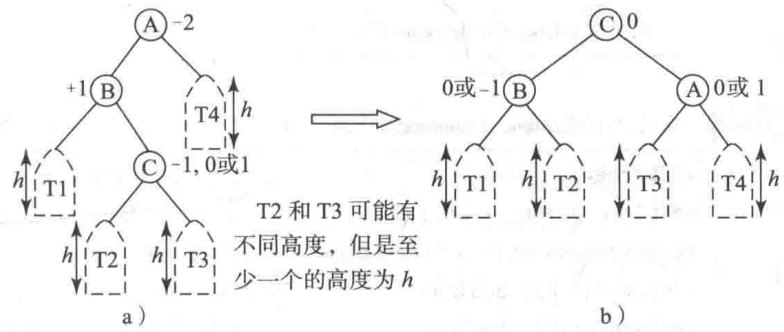


图 26-4 一个 LR 旋转修复 LR 不平衡

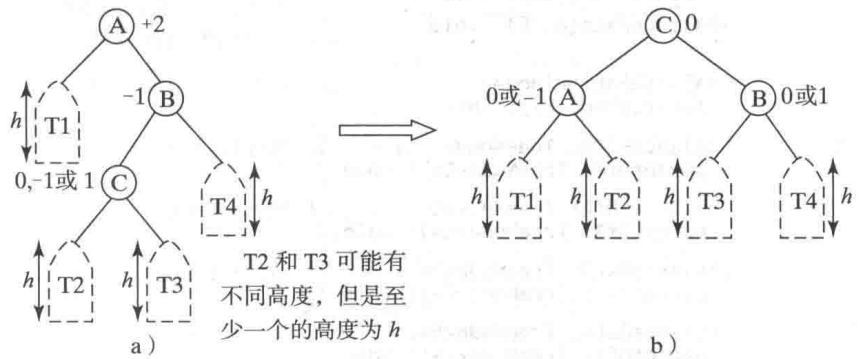


图 26-5 一个 RL 旋转修复 RL 不平衡

复习题

- 26.1 什么是 AVL 树？描述下面的词汇：平衡因子、左偏重、右偏重
- 26.2 给出如图 26-6 中所示树的每个结点的平衡因子。
- 26.3 描述一个 AVL 树的 LL 旋转、RR 旋转、LR 旋转以及 RL 旋转。

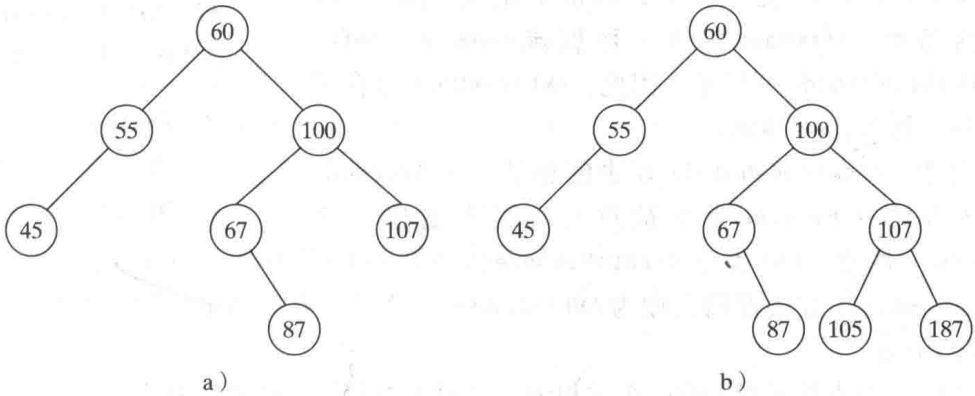


图 26-6 平衡因子决定一个结点是否平衡

26.3 为 AVL 树设计类

要点提示：由于 AVL 树是二叉查找树，AVLTree 设计为 BST 的子类。
由于 AVL 树是二叉查找树，可以继承 BST 类来定义 AVLTree 类，如图 26-7 所示。BST 和 TreeNode 类在 25.2.5 节中定义。

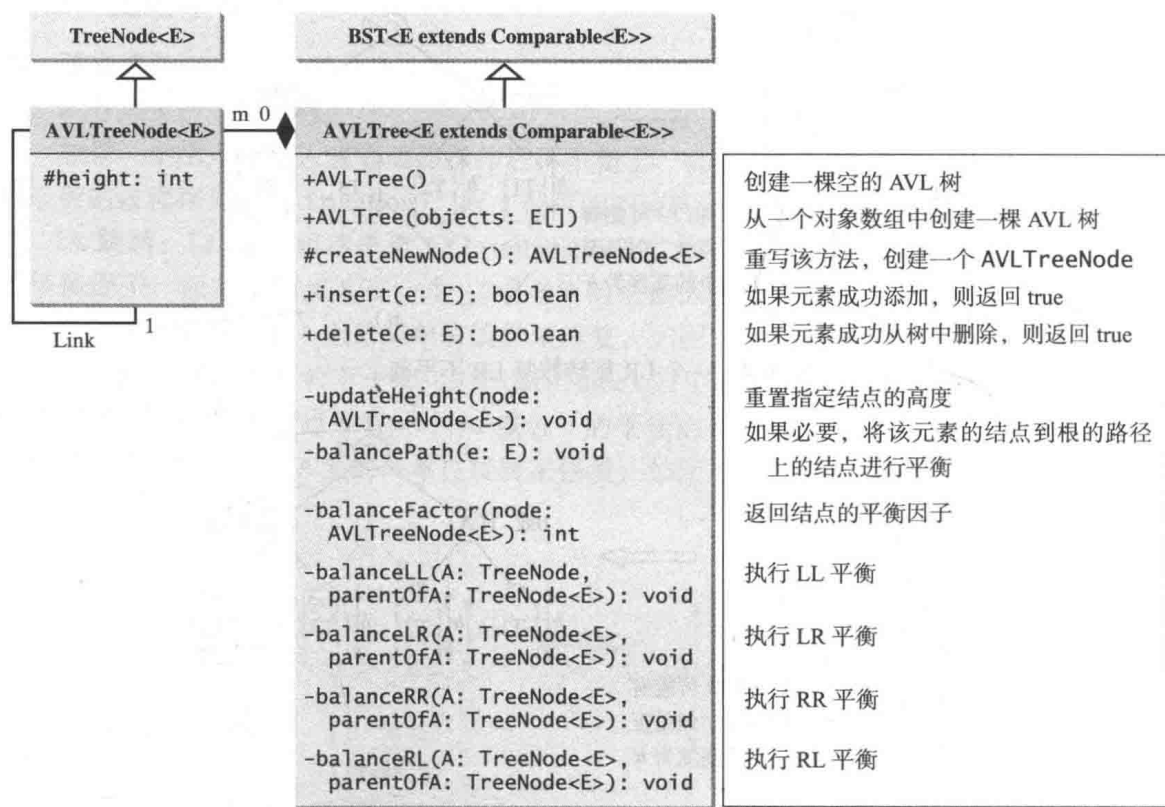


图 26-7 AVLTree 类继承自 BST，为 insert 和 delete 方法提供了新的实现

为了平衡一棵树，需要知道每个结点的高度。为了方便起见，保存每个结点的高度在 AVLTreeNode 中，并定义 AVLTreeNode 为 BST.TreeNode 的子类。注意 TreeNode 定义为 BST 中的静态内部类。AVLTreeNode 将定义为 AVLTree 的静态内部类。TreeNode 包含了数据域 element、left、right，被 AVLTreeNode 所继承。因此，AVLTreeNode 包含了 4 个数据域，如图 26-8 所示。

BST 类中，createNewNode() 方法创建了一个 TreeNode 对象。该方法在 AVLTree 类中被重写，用于创建一个 AVLTreeNode。注意，BST 类中 createNewNode() 方法返回类型为 TreeNode，而 AVLTree 类中 createNewNode() 方法返回类型为 AVLTreeNode。这是没有问题的，因为 AVLTreeNode 是 TreeNode 的子类。

在 AVLTree 中查找元素和在一个常规的二叉树中查找是一样的，因此定义在 BST 类中的 search 方法同样可以应用于 AVLTree。

insert 和 delete 方法被重写，用于插入和删除一个元素，必要的时候执行重新平衡的操作，从而保证树是平衡的。

复习题

- 26.4 AVLTreeNode 的数据域是什么？
- 26.5 真或者假：AVLTreeNode 是 TreeNode 的子类？
- 26.6 真或者假：AVLTree 是 BST 的子类。

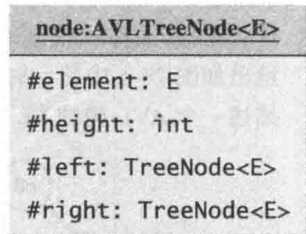


图 26-8 AVLTreeNode 包含保护类型数据域 element、height、left、right

26.4 重写 insert 方法

🔑 **要点提示：**插入一个元素到 AVL 树中和插入到 BST 中是一样的，不同之处在于树可能需要重新平衡。

一个新的元素经常作为叶子结点被插入。作为增加一个新结点的结果，新的叶子结点的祖先结点的高度会增加。插入一个新的结点后，检查沿着新的叶子结点到根结点的路径上的结点，如果发现一个不平衡的结点，则使用程序清单 26-1 中的算法执行适当的旋转（见图 26-9）。

程序清单 26-1 平衡一条路径上的结点

```

1 balancePath(E e) {
2   Get the path from the node that contains element e to the root,
3   as illustrated in Figure 26.9;
4   for each node A in the path leading to the root {
5     Update the height of A;
6     Let parentOfA denote the parent of A,
7     which is the next node in the path, or null if A is the root;
8
9     switch (balanceFactor(A)) {
10      case -2: if balanceFactor(A.left) == -1 or 0
11               Perform LL rotation; // See Figure 26.2
12             else
13               Perform LR rotation; // See Figure 26.4
14             break;
15      case +2: if balanceFactor(A.right) == +1 or 0
16               Perform RR rotation; // See Figure 26.3
17             else
18               Perform RL rotation; // See Figure 26.5
19    } // End of switch
20  } // End of for
21 } // End of method

```

算法考察从新的叶子结点到根结点的每个结点。更新路径上的结点的高度。如果结点是平衡的，则无须执行任何动作。如果结点是不平衡的，则执行适当的旋转操作。

✓ 复习题

26.7 针对图 26-6a 中的 AVL 树，显示添加元素 40 之后的新的 AVL 树。为了重新平衡该树，需要执行什么旋转操作？哪个结点是不平衡的？

26.8 针对图 26-6a 中的 AVL 树，显示添加元素 50 之后的新的 AVL 树。为了重新平衡该树，需要执行什么旋转操作？哪个结点是不平衡的？

26.9 针对图 26-6a 中的 AVL 树，显示添加元素 80 之后的新的 AVL 树。为了重新平衡该树，需要执行什么旋转操作？哪个结点是不平衡的？

26.8 针对图 26-6a 中的 AVL 树，显示添加元素 89 之后的新的 AVL 树。为了重新平衡该树，需要执行什么旋转操作？哪个结点是不平衡的？

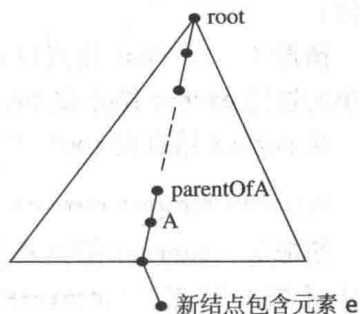


图 26-9 从新的叶子结点到根结点的路径上的结点可能变得不平衡

26.5 实现旋转

🔑 **要点提示：**通过执行适当的旋转操作，一个不平衡的树变得平衡。

第 26.2 节演示了如何在一个结点上执行旋转操作。程序清单 26-2 给出了 LL 旋转的算

法, 如图 26-2 所示。

程序清单 26-2 LL 旋转算法

```

1  balanceLL(TreeNode A, TreeNode parentOfA) {
2      Let B be the left child of A.
3
4      if (A is the root)
5          Let B be the new root
6      else {
7          if (A is a left child of parentOfA)
8              Let B be a left child of parentOfA;
9          else
10             Let B be a right child of parentOfA;
11     }
12
13     Make T2 the left subtree of A by assigning B.right to A.left;
14     Make A the right child of B by assigning A to B.right;
15     Update the height of node A and node B;
16 } // End of method

```

注意, 结点 A 和 B 的高度可以被改变, 而树中的其他结点的高度没有被改变。可以以相似的方式实现 RR、LR 以及 RL 旋转。

26.6 实现 delete 方法

要点提示: 从 AVL 树中删除一个元素和从 BST 中删除是一样的, 不同之处在于树可能需要重新平衡。

如 25.3 节所讨论的, 从一棵二叉树中删除一个元素, 算法首先定位包含元素的结点。让 `current` 指向二叉树中包含该元素的结点, `parent` 指向 `current` 结点的父结点。`current` 结点可能是 `parent` 结点的左子结点或者右子结点。当删除一个元素的时候, 可能出现两种情形:

情形 1: `current` 结点没有左子结点, 如图 25-10a 所示。为了删除 `current` 结点, 只需简单的连接 `parent` 结点和 `current` 结点的右子结点, 如图 25-10b 所示。

从 `parent` 结点到 `root` 结点路径上的结点的高度可能减少。为了保证树是平衡的, 调用

```
balancePath(parent.element); // Defined in Listing 26.1
```

情形 2: `current` 结点具有左子结点。让 `rightMost` 指向 `current` 结点的左子树中包含最大元素的结点, `parentOfRightMost` 指向 `rightMost` 结点的父结点, 如图 25-12a 所示。`rightMost` 结点不会有右子结点, 但是可能有左子结点。替换 `current` 结点中的元素值为 `rightMost` 结点中的值, 并连接 `parentOfRightMost` 结点和 `rightMost` 结点的左子结点, 删除 `rightMost` 结点, 如图 25-12b 所示。

从 `parentOfRightMost` 结点到根结点路径上的结点的高度可能减少。为了保证树是平衡的, 调用

```
balancePath(parentOfRightMost); // Defined in Listing 26.1
```

复习题

26.11 针对图 26-6a 中的 AVL 树, 显示删除元素 107 之后的新的 AVL 树。为了重新平衡该树, 需要执行什么旋转操作? 哪个结点是不平衡的?


26.11 针对图 26-6a 中的 AVL 树, 显示删除元素 60 之后的新的 AVL 树。为了重新平衡该树, 需要执

行什么旋转操作？哪个结点是不平衡的？

26.11 针对图 26-6a 中的 AVL 树，显示删除元素 55 之后的新的 AVL 树。为了重新平衡该树，需要执行什么旋转操作？哪个结点是不平衡的？

26.11 针对图 26-6b 中的 AVL 树，显示删除元素 67 和 87 之后的新的 AVL 树。为了重新平衡该树，需要执行什么旋转操作？哪个结点是不平衡的？

26.7 AVLTree 类

 **要点提示：** AVLTree 类继承自 BST 类，重写 insert 和 delete 方法，从而必要的时候重新平衡该树。

程序清单 26-3 给出了 AVLTree 类的完整源代码。

程序清单 26-3 AVLTree.java

```

1 public class AVLTree<E> extends Comparable<E>> extends BST<E> {
2     /** Create an empty AVL tree */
3     public AVLTree() {
4     }
5
6     /** Create an AVL tree from an array of objects */
7     public AVLTree(E[] objects) {
8         super(objects);
9     }
10
11     @Override /** Override createNewNode to create an AVLTreeNode */
12     protected AVLTreeNode<E> createNewNode(E e) {
13         return new AVLTreeNode<E>(e);
14     }
15
16     @Override /** Insert an element and rebalance if necessary */
17     public boolean insert(E e) {
18         boolean successful = super.insert(e);
19         if (!successful)
20             return false; // e is already in the tree
21         else {
22             balancePath(e); // Balance from e to the root if necessary
23         }
24
25         return true; // e is inserted
26     }
27
28     /** Update the height of a specified node */
29     private void updateHeight(AVLTreeNode<E> node) {
30         if (node.left == null && node.right == null) // node is a leaf
31             node.height = 0;
32         else if (node.left == null) // node has no left subtree
33             node.height = 1 + ((AVLTreeNode<E>)(node.right)).height;
34         else if (node.right == null) // node has no right subtree
35             node.height = 1 + ((AVLTreeNode<E>)(node.left)).height;
36         else
37             node.height = 1 +
38                 Math.max(((AVLTreeNode<E>)(node.right)).height,
39                     ((AVLTreeNode<E>)(node.left)).height);
40     }
41
42     /** Balance the nodes in the path from the specified
43      * node to the root if necessary
44      */
45     private void balancePath(E e) {

```

```

46 java.util.ArrayList<TreeNode<E>> path = path(e);
47 for (int i = path.size() - 1; i >= 0; i--) {
48     AVLTreeNode<E> A = (AVLTreeNode<E>)(path.get(i));
49     updateHeight(A);
50     AVLTreeNode<E> parentOfA = (A == root) ? null :
51         (AVLTreeNode<E>)(path.get(i - 1));
52
53     switch (balanceFactor(A)) {
54     case -2:
55         if (balanceFactor((AVLTreeNode<E>)A.left) <= 0) {
56             balanceLL(A, parentOfA); // Perform LL rotation
57         }
58         else {
59             balanceLR(A, parentOfA); // Perform LR rotation
60         }
61         break;
62     case +2:
63         if (balanceFactor((AVLTreeNode<E>)A.right) >= 0) {
64             balanceRR(A, parentOfA); // Perform RR rotation
65         }
66         else {
67             balanceRL(A, parentOfA); // Perform RL rotation
68         }
69     }
70 }
71 }
72
73 /** Return the balance factor of the node */
74 private int balanceFactor(AVLTreeNode<E> node) {
75     if (node.right == null) // node has no right subtree
76         return -node.height;
77     else if (node.left == null) // node has no left subtree
78         return +node.height;
79     else
80         return ((AVLTreeNode<E>)node.right).height -
81             ((AVLTreeNode<E>)node.left).height;
82 }
83
84 /** Balance LL (see Figure 26.2) */
85 private void balanceLL(TreeNode<E> A, TreeNode<E> parentOfA) {
86     TreeNode<E> B = A.left; // A is left-heavy and B is left-heavy
87
88     if (A == root) {
89         root = B;
90     }
91     else {
92         if (parentOfA.left == A) {
93             parentOfA.left = B;
94         }
95         else {
96             parentOfA.right = B;
97         }
98     }
99
100     A.left = B.right; // Make T2 the left subtree of A
101     B.right = A; // Make A the left child of B
102     updateHeight((AVLTreeNode<E>)A);
103     updateHeight((AVLTreeNode<E>)B);
104 }
105
106 /** Balance LR (see Figure 26.4) */
107 private void balanceLR(TreeNode<E> A, TreeNode<E> parentOfA) {
108     TreeNode<E> B = A.left; // A is left-heavy
109     TreeNode<E> C = B.right; // B is right-heavy

```

```

110
111     if (A == root) {
112         root = C;
113     }
114     else {
115         if (parentOfA.left == A) {
116             parentOfA.left = C;
117         }
118         else {
119             parentOfA.right = C;
120         }
121     }
122
123     A.left = C.right; // Make T3 the left subtree of A
124     B.right = C.left; // Make T2 the right subtree of B
125     C.left = B;
126     C.right = A;
127
128     // Adjust heights
129     updateHeight((AVLTreeNode<E>)A);
130     updateHeight((AVLTreeNode<E>)B);
131     updateHeight((AVLTreeNode<E>)C);
132 }
133
134 /** Balance RR (see Figure 26.3) */
135 private void balanceRR(TreeNode<E> A, TreeNode<E> parentOfA) {
136     TreeNode<E> B = A.right; // A is right-heavy and B is right-heavy
137
138     if (A == root) {
139         root = B;
140     }
141     else {
142         if (parentOfA.left == A) {
143             parentOfA.left = B;
144         }
145         else {
146             parentOfA.right = B;
147         }
148     }
149
150     A.right = B.left; // Make T2 the right subtree of A
151     B.left = A;
152     updateHeight((AVLTreeNode<E>)A);
153     updateHeight((AVLTreeNode<E>)B);
154 }
155
156 /** Balance RL (see Figure 26.5) */
157 private void balanceRL(TreeNode<E> A, TreeNode<E> parentOfA) {
158     TreeNode<E> B = A.right; // A is right-heavy
159     TreeNode<E> C = B.left; // B is left-heavy
160
161     if (A == root) {
162         root = C;
163     }
164     else {
165         if (parentOfA.left == A) {
166             parentOfA.left = C;
167         }
168         else {
169             parentOfA.right = C;
170         }
171     }
172 }

```

```

173     A.right = C.left; // Make T2 the right subtree of A
174     B.left = C.right; // Make T3 the left subtree of B
175     C.left = A;
176     C.right = B;
177
178     // Adjust heights
179     updateHeight((AVLTreeNode<E>)A);
180     updateHeight((AVLTreeNode<E>)B);
181     updateHeight((AVLTreeNode<E>)C);
182 }
183
184 @Override /** Delete an element from the AVL tree.
185  * Return true if the element is deleted successfully
186  * Return false if the element is not in the tree */
187 public boolean delete(E element) {
188     if (root == null)
189         return false; // Element is not in the tree
190
191     // Locate the node to be deleted and also locate its parent node
192     TreeNode<E> parent = null;
193     TreeNode<E> current = root;
194     while (current != null) {
195         if (element.compareTo(current.element) < 0) {
196             parent = current;
197             current = current.left;
198         }
199         else if (element.compareTo(current.element) > 0) {
200             parent = current;
201             current = current.right;
202         }
203         else
204             break; // Element is in the tree pointed by current
205     }
206
207     if (current == null)
208         return false; // Element is not in the tree
209
210     // Case 1: current has no left children (See Figure 25.10)
211     if (current.left == null) {
212         // Connect the parent with the right child of the current node
213         if (parent == null) {
214             root = current.right;
215         }
216         else {
217             if (element.compareTo(parent.element) < 0)
218                 parent.left = current.right;
219             else
220                 parent.right = current.right;
221
222             // Balance the tree if necessary
223             balancePath(parent.element);
224         }
225     }
226     else {
227         // Case 2: The current node has a left child
228         // Locate the rightmost node in the left subtree of
229         // the current node and also its parent
230         TreeNode<E> parentOfRightMost = current;
231         TreeNode<E> rightMost = current.left;
232
233         while (rightMost.right != null) {
234             parentOfRightMost = rightMost;
235             rightMost = rightMost.right; // Keep going to the right
236         }

```

```

238         // Replace the element in current by the element in rightMost
239         current.element = rightMost.element;
240
241         // Eliminate rightmost node
242         if (parentOfRightMost.right == rightMost)
243             parentOfRightMost.right = rightMost.left;
244         else
245             // Special case: parentOfRightMost is current
246             parentOfRightMost.left = rightMost.left;
247
248         // Balance the tree if necessary
249         balancePath(parentOfRightMost.element);
250     }
251
252     size--;
253     return true; // Element inserted.
254 }
255
256 /** AVLTreeNode is TreeNode plus height */
257 protected static class AVLTreeNode<E extends Comparable<E>>
258     extends BST.TreeNode<E> {
259     protected int height = 0; // New data field
260
261     public AVLTreeNode(E e) {
262         super(e);
263     }
264 }
265 }

```

AVLTree 类继承 BST。和 BST 类一样，AVLTree 类具有一个无参的构造方法，用于构建一个空的 AVLTree（第 3 ~ 4 行），以及一个从一个元素数组构建一个初始化的 AVLTree（第 7 ~ 9 行）。

定义在 BST 类中的 createNewNode() 方法创建一个 TreeNode。这个方法被重写以返回一个 AVLTreeNode（第 12 ~ 14 行）。

AVLTree 中的 insert 方法在第 17 ~ 26 行重写。该方法首先调用 BST 中的 insert 方法，然后调用 balancePath(e)（第 22 行）来保证树是平衡的。

balancePath 方法首先得到从包含元素 e 的结点到根结点的路径上的所有结点（第 46 行）。对于路径上的每个结点，更新它的高度（第 49 行），检查它的平衡因子（第 53 行），如果必要的话执行适当的旋转（第 53 ~ 69 行）。

执行旋转的 4 个方法在第 85 ~ 182 行定义。每个方法带两个 TreeNode 类型的参数——A 和 parentOfA——来执行结点 A 处的适当的旋转。每个旋转是如何执行的在图 26-2 ~ 图 26-5 中展示。旋转后，结点 A、B 以及 C 的高度被更新（第 102、129、152 和 179 行）。

AVLTree 中的 delete 方法在第 187 ~ 264 行被重写。该方法和 BST 类中实现的一样，不同之处在于需要在删除之后的两种情形中重新平衡结点（第 224 和 249 行）。

✓ 复习题

- 26.15 为什么 createNewNode 方法定义为受保护的？
- 26.16 updateHeight 方法什么时候调用？balanceFactor 方法什么时候调用？balancePath 方法什么时候调用？
- 26.17 AVLTree 类中的数据域是什么？
- 26.18 insert 和 delete 方法中，一旦执行了一个旋转来平衡树中的结点，那么可能还有不平衡的结点吗？

26.8 测试 AVLTree 类

🔑 **要点提示：**本节给出一个使用 AVLTree 类的例子。

程序清单 26-4 给出了一个测试程序。程序创建了一个 AVLTree，使用整数数组 25、20 和 5 来进行初始化（第 4～5 行），在第 9～18 行插入元素，第 22～28 行删除元素。由于 AVLTree 是 BST 的子类，而 BST 中的元素是可以遍历的，程序第 33～35 行使用了 foreach 循环来遍历所有的元素。

程序清单 26-4 TestAVLTree.java

```

1 public class TestAVLTree {
2     public static void main(String[] args) {
3         // Create an AVL tree
4         AVLTree<Integer> tree = new AVLTree<>(new Integer[]{25,
5             20, 5});
6         System.out.print("After inserting 25, 20, 5:");
7         printTree(tree);
8
9         tree.insert(34);
10        tree.insert(50);
11        System.out.print("\nAfter inserting 34, 50:");
12        printTree(tree);
13
14        tree.insert(30);
15        System.out.print("\nAfter inserting 30");
16        printTree(tree);
17
18        tree.insert(10);
19        System.out.print("\nAfter inserting 10");
20        printTree(tree);
21
22        tree.delete(34);
23        tree.delete(30);
24        tree.delete(50);
25        System.out.print("\nAfter removing 34, 30, 50:");
26        printTree(tree);
27
28        tree.delete(5);
29        System.out.print("\nAfter removing 5:");
30        printTree(tree);
31
32        System.out.print("\nTraverse the elements in the tree: ");
33        for (int e: tree) {
34            System.out.print(e + " ");
35        }
36    }
37
38    public static void printTree(BST tree) {
39        // Traverse tree
40        System.out.print("\nInorder (sorted): ");
41        tree.inorder();
42        System.out.print("\nPostorder: ");
43        tree.postorder();
44        System.out.print("\nPreorder: ");
45        tree.preorder();
46        System.out.print("\nThe number of nodes is " + tree.getSize());
47        System.out.println();
48    }
49 }

```

After inserting 25, 20, 5:
Inorder (sorted): 5 20 25

```
Postorder: 5 25 20
Preorder: 20 5 25
The number of nodes is 3

After inserting 34, 50:
Inorder (sorted): 5 20 25 34 50
Postorder: 5 25 50 34 20
Preorder: 20 5 34 25 50
The number of nodes is 5

After inserting 30
Inorder (sorted): 5 20 25 30 34 50
Postorder: 5 20 30 50 34 25
Preorder: 25 20 5 34 30 50
The number of nodes is 6

After inserting 10
Inorder (sorted): 5 10 20 25 30 34 50
Postorder: 5 20 10 30 50 34 25
Preorder: 25 10 5 20 34 30 50
The number of nodes is 7

After removing 34, 30, 50:
Inorder (sorted): 5 10 20 25
Postorder: 5 20 25 10
Preorder: 10 5 25 20
The number of nodes is 4

After removing 5:
Inorder (sorted): 10 20 25
Postorder: 10 25 20
Preorder: 20 10 25
The number of nodes is 3
Traverse the elements in the tree: 10 20 25
```

图 26-10 显示了当元素添加到树上后, 树是如何演化的。25 和 20 添加后, 树如图 26-10a 所示。5 作为 20 的左子结点插入, 如图 26-10b 所示。树是不平衡的, 在结点 25 处是左偏重的。执行一个 LL 操作来获得一棵 AVL 树, 如图 26-10c 所示。

插入 34 后, 树如图 26-10d 所示。插入 50 后, 树如图 26-10e 所示。树是不平衡的, 在结点 25 处是右偏重的。执行一个 RR 操作来获得一棵 AVL 树, 如图 26-10f 所示。

插入 30 后, 树如图 26-10g 所示。树是不平衡的。执行一个 RL 操作来获得一棵 AVL 树, 如图 26-10h 所示。

插入 10 后, 树如图 26-10i 所示。树是不平衡的。执行一个 LR 操作来获得一棵 AVL 树, 如图 26-10j 所示。

图 26-11 显示了当元素被删除后树是如何演化的。删除 34、30 以及 50 后, 树如图 26-11b 所示。树不是平衡的。执行一个 LL 旋转来得到一棵 AVL 树, 如图 26-11c 所示。

删除 5 后, 树如图 26-11d 所示。树不是平衡的。执行一个 RL 旋转来得到一棵 AVL 树, 如图 26-11e 所示。

✓ 复习题

26.19 顺序插入 1, 2, 3, 4, 10, 9, 7, 5, 8, 6 到树中后, 显示 AVL 树的变化。

26.20 针对前面问题所构建的树, 顺序删除 1, 2, 3, 4, 10, 9, 7, 5, 8, 6 后, 显示它的变化。

26.21 可以使用 foreach 循环来遍历 AVL 树中的元素吗?

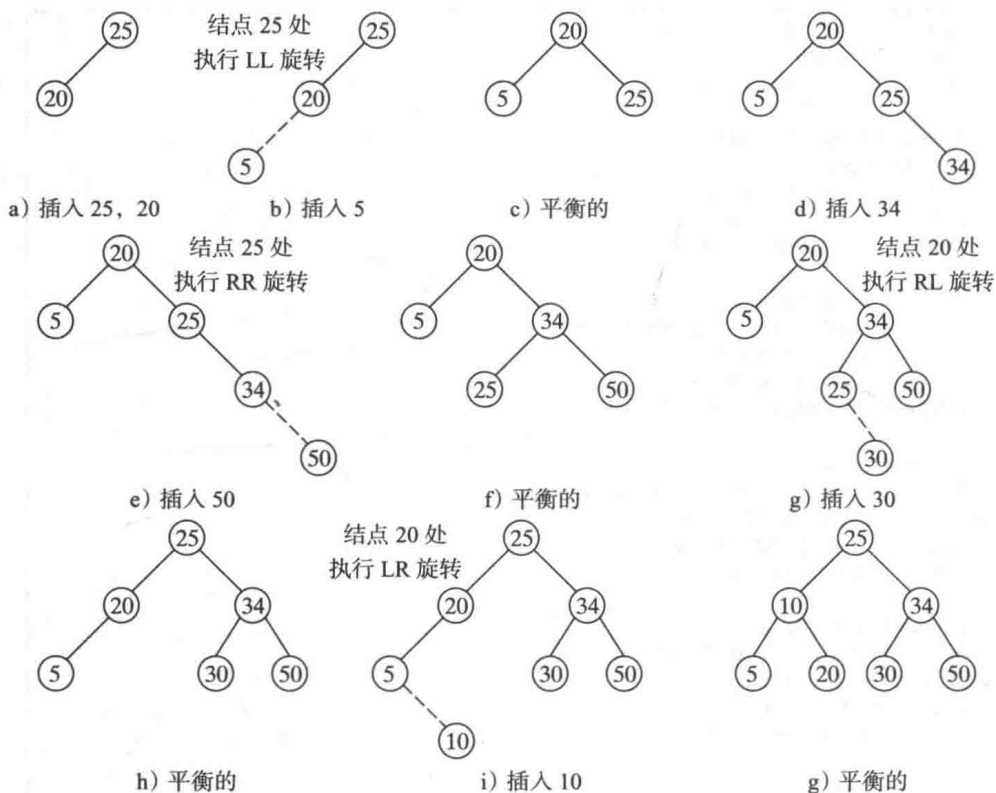


图 26-10 新的元素插入后树的演化

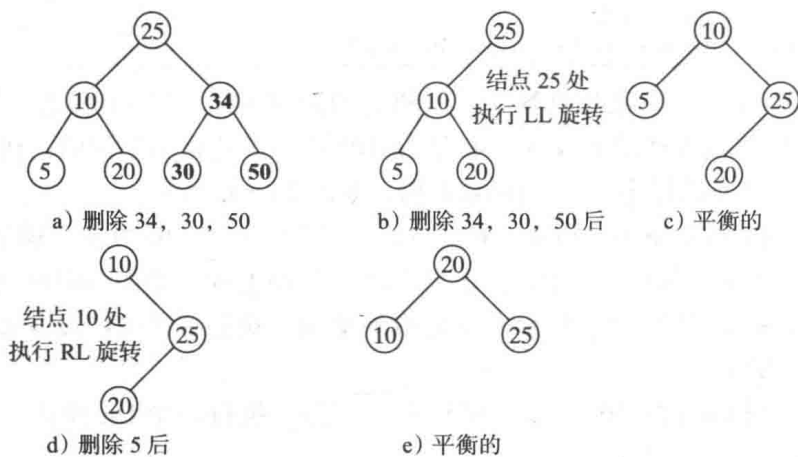


图 26-11 当元素从树中删除后树的演化

26.9 AVL 树的时间复杂度分析

要点提示: 由于一个 AVL 树的高度为 $O(\log n)$, 所以 AVLTree 树中的 search、insert 以及 delete 方法的时间复杂度为 $O(\log n)$ 。

AVLTree 中的 search、insert 以及 delete 方法的时间复杂度依赖于树的高度。我们可以证明树的高度为 $O(\log n)$ 。

使用 $G(h)$ 表示一个具有高度为 h 的 AVL 树的最小结点数。显然, $G(1)$ 为 1, $G(2)$ 为 2。具有高度为 $h \geq 3$ 的 AVL 树的最小结点数会有两个最小子树: 一个具有高度 $h-1$, 另外一

个具有高度 $h-2$ 。因此,

$$G(h) = G(h-1) + G(h-2) + 1$$

回顾下, 在下标为 i 处的斐波那契数字可以使用递推关系 $F(i) = F(i-1) + F(i-2)$ 。因此, 函数 $G(h)$ 实质上与 $F(i)$ 一样。可以证明

$$h < 1.4405 \log(n+2) - 1.3277$$

这里 n 是树中的结点数。因此, 一个 AVL 树的高度为 $O(\log n)$ 。

`search`、`insert` 以及 `delete` 方法只涉及树中沿着一条路径上的结点。`updateHeight` 和 `balanceFactor` 方法对于路径上的结点来说执行常量时间。`balance Path` 方法对于路径上的结点来说也执行常量时间。因此, `search`、`insert` 以及 `delete` 方法的时间复杂度为 $O(\log n)$ 。

复习题

- 26.22 对于具有 3 个结点、5 个结点以及 7 个结点的 AVL 树来说, 最大 / 最小高度是多少?
- 26.23 如果一棵 AVL 树高度为 3, 该树可以具有的最大结点数目为多少? 该树可以具有的最小结点数目为多少?
- 26.24 如果一棵 AVL 树高度为 4, 该树可以具有的最大结点数目为多少? 该树可以具有的最小结点数目为多少?

关键术语

AVL tree (高度平衡二叉树)	right-heavy (右偏重)
balance factor (平衡因子)	RL rotation (RL 旋转)
left-heavy (左偏重)	rotation (旋转)
LL rotation (LL 旋转)	RR rotation (RR 旋转)
LR rotation (LR 旋转)	well-balanced tree (高度平衡树)
perfectly balanced tree (完全平衡树)	

本章小结

1. AVL 树是高度平衡二叉树。在一棵 AVL 树中, 每个结点的两个子树的高度差为 0 或者 1。
2. 在一棵 AVL 树中插入或者删除元素的过程和在常规的二叉查找树中是一样的。不同之处在于可能需要在插入或者删除后重新平衡该树。
3. 插入和删除引起的树的不平衡, 通过不平衡结点处的子树的旋转重新获得平衡。
4. 一个结点的重新平衡的过程称为旋转。有 4 种可能的旋转: LL 旋转、LR 旋转、RR 旋转、RL 旋转。
5. AVL 树的高度为 $O(\log n)$ 。因此, `search`、`insert` 以及 `delete` 方法的时间复杂度为 $O(\log n)$ 。

测试题

回答位于网址 www.cs.armstrong.edu/liang/intro10e/quiz.html 的本章测试题。

编程练习题

- *26.1 (图形化显示 AVL 树) 编写一个程序, 显示一棵 AVL 树, 每个结点处显示平衡因子。
- 26.2 (比较性能) 编写一个测试程序, 随机产生 500 000 个数字, 并将它们插入 BST 中, 然后重新打乱这 500 000 个数字然后执行一次查找, 再次打乱数字并从树中删除。编写另外一个测试程序, 为 AVLTree 执行同样的操作。比较两个程序的执行时间。
- ***26.3 (AVL 树的动画) 编写一个程序, 实现 AVL 树的 `insert`、`delete` 以及 `search` 方法的动画, 如

图 26-1 所示。

****26.4** (BST 的父引用) 假定定义在 BST 中的 `TreeNode` 类包含了指向结点的父结点的引用, 如编程练习题 25.15 所示。实现 `AVLTree` 类来支持这个改变。编写一个测试程序, 添加数字 1, 2, ..., 100 到该树中并显示所有叶子结点的路径。

****26.5** (第 k 小的元素) 可以通过中序遍历在 $O(n)$ 的时间内找到 BST 中第 k 小的元素。对于一棵 AVL 树而言, 可以在 $O(\log n)$ 时间内找到。为了做到这点, 在 `AVLTreeNode` 中添加一个命名为 `size` 的新的数据域, 存储以该结点为根结点的子树的结点数。注意, 一个结点 v 比它的两个子结点的大小的和多 1。图 26-12 显示了一棵 AVL 树, 以及树中每个结点的 `size` 值。

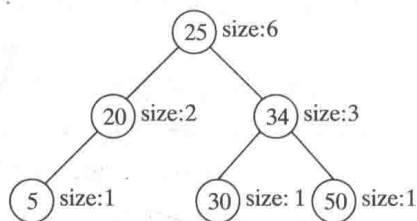


图 26-12 `AVLTreeNode` 中的 `size` 数据域存储以该结点为根结点的子树的结点数

`AVLTree` 类中, 添加以下方法, 返回树中第 k 小的元素。

```
public E find(int k)
```

如果 $k < 1$ 或者 $k > \text{the size of the tree}$ (树的大小), 方法返回 `null`。使用递归方法 `find(k, root)` 实现该方法, 递归方法返回以指定根元素的树中的第 k 小的元素。让 A 和 B 作为该根元素的左子结点和右子结点。假设树不为空, 并且 $k \leq \text{root.size}$, 可以如下递归定义 `find(k, root)`:

$$\text{find}(k, \text{root}) = \begin{cases} \text{root.element, if } A \text{ is null and } k \text{ is } 1; \\ B.\text{element, if } A \text{ is null and } k \text{ is } 2; \\ \text{find}(k, A), \text{ if } k \leq A.\text{size}; \\ \text{root.element, if } k = A.\text{size} + 1; \\ \text{find}(k - A.\text{size} - 1, B), \text{ if } k > A.\text{size} + 1; \end{cases}$$

修改 `AVLTree` 树中的 `insert` 和 `delete` 方法, 为每个结点中的 `size` 属性设置正确的值。`insert` 和 `delete` 方法仍然为 $O(\log n)$ 时间。`find(k)` 方法可以在 $O(\log n)$ 时间内执行。因此, 可以在一棵 AVL 树中在 $O(\log n)$ 时间内找到第 k 小的元素。

使用下面的主方法来测试你的程序:

```
import java.util.Scanner;
```

```
public class Exercise26_05 {
    public static void main(String[] args) {
        AVLTree<Double> tree = new AVLTree<>();
        Scanner input = new Scanner(System.in);
        System.out.print("Enter 15 numbers: ");
        for (int i = 0; i < 15; i++) {
            tree.insert(input.nextDouble());
        }

        System.out.print("Enter k: ");
        System.out.println("The " + k + "th smallest number is " +
            tree.find(k));
    }
}
```

****26.6** (测试 `AVLTree`) 设计和编写一个完整的测试程序, 测试程序清单 26-4 中的 `AVLTree` 类是否满足所有要求。

散 列

【】 教学目标

- 理解什么是散列，以及散列用于什么（27.2 节）。
- 获得一个对象的散列码，设计一个散列函数，将一个键映射到一个索引（27.3 节）。
- 使用公开地址处理冲突（27.4 节）。
- 了解线性探测、二次探测和再哈希法的区别（27.4 节）。
- 使用链地址法处理冲突（27.5 节）。
- 理解装填因子以及再散列（27.6 节）。
- 使用散列实现 MyHashMap（27.7 节）。
- 使用散列实现 MyHashSet（27.8 节）。

27.1 引言

🔑 要点提示：散列非常高效。使用散列将耗费 $O(1)$ 时间来查找、插入以及删除一个元素。

前面章节介绍了二叉查找树。在一个良好平衡的查找树中，可以在 $O(\log n)$ 时间内找到一个元素。还有更加高效的方法在一个容器中查找一个元素吗？本章介绍一种称为散列（hashing）的技术。可以使用散列来实现一个映射表或者集合，从而在 $O(1)$ 时间内查找、插入以及删除一个元素。

27.2 什么是散列

🔑 要点提示：散列使用一个散列函数，将一个键映射到一个索引上。

介绍散列之前，让我们回顾下映射表。映射表是一种使用散列实现的数据结构。回顾下映射表（21.5 节中介绍）是一种存储条目的容器对象。每个条目包含两部分：一个键（key）和一个值（value）。键又称为搜索键，用于查找相应的值。例如，一个字典可以存储在一个映射表中，其中单词作为键，而单词的定义作为值。

【】 注意：映射表（map）又称为字典（dictionary）、散列表（hash table）或者关联数组（associate array）。

Java 合集框架定义了 `java.util.Map` 接口来对映射表建模。三个具体的实现类为 `java.util.HashMap`、`java.util.LinkedHashMap` 以及 `java.util.TreeMap`。`java.util.HashMap` 使用散列实现，`java.util.LinkedHashMap` 使用 `LinkedList`，`java.util.TreeMap` 使用红黑树（奖励章节 4.1 介绍红黑树）。本章中你将学习到散列的概念，并使用它来实现一个散列映射表。

如果知道一个数组中元素的索引，可以使用索引在 $O(1)$ 时间内获得元素。因此这是否意味着我们可以将值存储在一个数组中，然后是用键作为索引来找到值呢？答案是可以的——如果可以将键映射到一个索引上。存储了值的数组称为散列表（hash table）。将键映射到散列表中的索引上的函数称为散列函数（hash function）。如图 27-1 所示，散列函数从

一个键获得索引，并使用索引来获取该键的值。散列（hashing）是一种无须执行搜索，即可通过从键得到的索引来获取值的技术。

如何设计一个散列函数，从一个键得到一个索引呢？理想的，我们希望设计一个函数，将每个搜索的键映射到散列表中的不同索引上。这样的函数称为完美散列函数（perfect hash function）。然而，很难找到一个完美散列函数。当两个或者更多的键映射到一个散列值上的时候，我们称为产生了一个冲突（collision）。尽管有办法来处理冲突，这将在本章后面进行讨论，但是最好首先就避免发生冲突。因此，应该设计一个快速以及易于计算的散列函数，来最小化冲突。

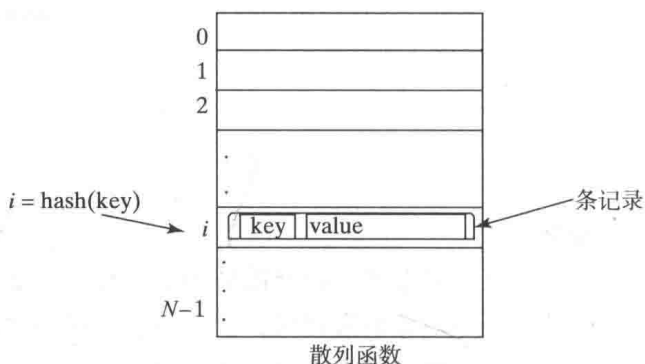


图 27-1 散列函数将键映射到散列表中的索引上

复习题

27.1 什么是散列函数？什么是完美散列函数？什么是冲突？

27.3 散列函数和散列码

要点提示：典型的散列函数首先将搜索键转换成一个称为散列码的整数值，然后将散列码压缩为散列表中的索引。

Java 的根类 Object 具有 hashCode 方法，该方法返回一个整数的散列码。默认的，该方法返回一个该对象的内存地址。hashCode 方法的一般约定如下：

- 1) 当 equals 方法被重写时，应该重写 hashCode 方法，从而保证两个相等的对象返回同样的散列码。
- 2) 程序执行中，如果对象的数据没有被修改，则多次调用 hashCode 将返回同样的整数。
- 3) 两个不相等的对象可能具有同样的散列码，但是应该在实现 hashCode 方法时避免太多这样的情形出现。

27.3.1 基本数据类型的散列码

对于 byte、short、int 以及 char 类型的搜索键而言，简单地将它们转型为 int。因此，这些类型中的任何一个的不同搜索键将有不同的散列码。

对于 float 类型的搜索键，使用 Float.floatToIntBits(key) 作为散列码。注意，floatToIntBits(float f) 返回一个 int 值，该值的比特表示和浮点数 f 的比特表示相同。因此，两个不同的 float 类型的搜索键将具有不同的散列码。

对于 long 类型的搜索键，简单地将类型转换为 int 将不是很好的选择，因为所有前 32 比特不同的键将具有相同的散列码。考虑到前 32 比特，将 64 比特分为两部分，并执行异或操作将两部分结合。这个过程称为折叠（folding）。一个 long 类型键的散列码为：

```
int hashCode = (int)(key ^ (key >> 32));
```

注意，>> 为右移操作符，将比特向右移动 32 位。例如，1010110 >> 2 得到 0010101。
^ 是比特异或操作。它在双目操作数的相应比特位上执行操作。例如，1010110^0110111 得

到 1100001。对于更多的比特操作，参加附录 G。

对于 `double` 类型的搜索键，首先使用 `Double.doubleToLongBits` 方法转化为 `long` 值。然后如下执行折叠操作：

```
long bits = Double.doubleToLongBits(key);
int hashCode = (int)(bits ^ (bits >> 32));
```

27.3.2 字符串类型的散列码

搜索键经常是字符串，因此为字符串设计好的散列函数非常重要。一个比较直观的方法是将所有字符的 Unicode 求和作为字符串的散列码。这个方法在应用中的搜索键不包含同样字母的情况下可能可以工作，但是如果搜索键包含同样字母，将产生许多冲突，例如 `tod` 和 `dot`。

一个更好的方法是考虑字符的位置，然后产生散列码。具体的，让散列码为：

$$s_0 \times b^{(n-1)} + s_1 \times b^{(n-2)} + \cdots + s_{n-1}$$

这里 s_i 为 `s.charAt(i)`。这个表达式为一些正数 b 的多项式，因此被称为多项式散列码 (polynomial hash code)。使用针对多项式方程的 Horner 规则 (参见 6.7 节)，可以如下高效地计算散列码：

$$(\cdots ((s_0 \times b + s_1)b + s_2)b + \cdots + s_{n-2})b + s_{n-1}$$

这个计算对于长的字符串来说，会导致溢出，但是 Java 中会忽略算术溢出。应该选择一个合适的 b 值来最小化冲突。实验显示， b 的较好的取值为 31, 33, 37, 39 和 41。String 类中，hashCode 采用 b 值为 31 的多项式散列码计算被重写。

27.3.3 压缩散列码

键的散列码可能是一个很大的整数，超过了散列表索引的范围，因此需要将它缩小到适合索引的范围。假设散列表的索引处于 0 到 $N-1$ 之间。将一个整数缩小到 0 到 $N-1$ 之间的最通常的做法是使用

```
h(hashCode) = hashCode % N
```

保证索引均匀扩展，选择 N 为大于 2 的素数。

理想的，应该为 N 选择一个素数。然而，选择一个大的素数将很耗时。Java API 为 `java.util.HashMap` 的实现中， N 设置为一个 2 的幂值。这样的选择具有合理性。当 N 为 2 的幂值时，

```
h(hashCode) = hashCode % N
```

与下面式子一样

```
h(hashCode) = hashCode & (N - 1)
```

与号 `&` 是一个比特 AND 操作符 (参见附录 G)。如果两个比特都为 1，则两个相应的比特的 AND 操作结果为 1。例如，假设 $N=4$ ，以及 `hashCode = 11`， $11 \% 4 = 3$ ，这个与 `01011 & 00011 = 11` 相同。`&` 操作符比 `%` 操作符执行快许多。

为了保证散列码是均匀分布的，`java.util.HashMap` 的实现中采用了补充的散列函数与主散列函数一起使用。该函数定义为：

```
private static int supplementalHash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

\wedge 和 \gg 是比特的异或和无符号右移操作（也在附录 G 中介绍）。比特操作比乘、除，以及求余操作要快许多。应该尽量使用比特操作来代替这些操作。

完整的散列函数如下定义：

$h(\text{hashCode}) = \text{supplementalHash}(\text{hashCode}) \% N$

这个与以下式子一样：

$h(\text{hashCode}) = \text{supplementalHash}(\text{hashCode}) \& (N - 1)$

因为 N 是一个 2 的幂值。

复习题

- 27.2 什么是散列码？Byte、Short、Integer 以及 Character 的散列码为多少？
- 27.3 Float 对象的散列码是如何计算的？
- 27.4 Long 对象的散列码是如何计算的？
- 27.5 Double 对象的散列码是如何计算的？
- 27.6 String 对象的散列码是如何计算的？
- 27.7 一个散列码是如何压缩为一个表示散列表中索引的整数的？
- 27.8 如果 N 为 2 的幂值， $N/2$ 与 $N \gg 1$ 一样吗？
- 27.9 如果 N 为 2 的幂值，对于整数 m ， $m \% N$ 与 $m \& (N - 1)$ 等价吗？

27.4 使用开放地址法处理冲突

要点提示：当两个键映射到散列表中的同一个索引上，冲突发生。通常，有两种方法处理冲突：开放地址法和链地址法。

开放地址法（open addressing）是在冲突发生时，在散列表中找到一个开放位置的过程。开放地址法有几个变体：线性探测、二次探测和再哈希法。

27.4.1 线性探测

当插入一个条目到散列表中发生冲突时，线性探测法（linear probing）按顺序找到下一个可用的位置。例如，如果冲突发生在 $\text{hashTable}[k \% N]$ ，则检查 $\text{hashTable}[(k+1) \% N]$ 是否可用。如果不可用，则检查 $\text{hashTable}[(k+2) \% N]$ ，以此类推，直到一个可用单元被找到，如图 27-2 所示。

注意：当探测到表的终点时，则返回表的起点。因此，散列表被当成是循环的。

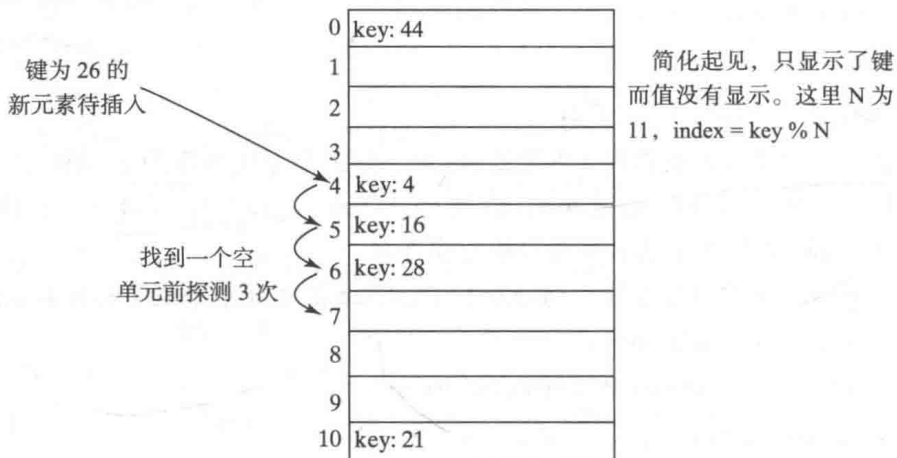


图 27-2 线性探测法按顺序找到下一个可用的位置

查找散列表中的条目时,从散列函数获得键相应的索引,比如说 k 。检查 $\text{hashTable}[k \% N]$ 是否包含该条目。如果没有,检查 $\text{hashTable}[(k+1) \% N]$ 是否包含该条目,以此类推,直到找到,或者达到一个空的单元。

删除散列表中的条目时,查找匹配键的条目。如果条目找到,则放置一个特殊的标记表示该条目是可用的。散列表中的每个单元具有三个可能的状态:被占的、标记的或者空的。注意,一个被标记的单元对于插入同样是可用的。

线性探测法容易导致散列表中连续的单元组被占用。每个组称为一个簇(cluster)。每个簇实际上成为在获取、添加以及删除一个条目时必须查找的探测序列。当簇的大小增加时,它们可能合并为更大的簇,从而更加放慢查找的时间。这是线性探测法的一个较大的缺点。

教学注意: 参见网址 www.cs.armstrong.edu/liang/animation/HashingLinearProbingAnimation.html, 获取线性探测法如何工作的交互式 GUI 演示,如图 27-3 所示。

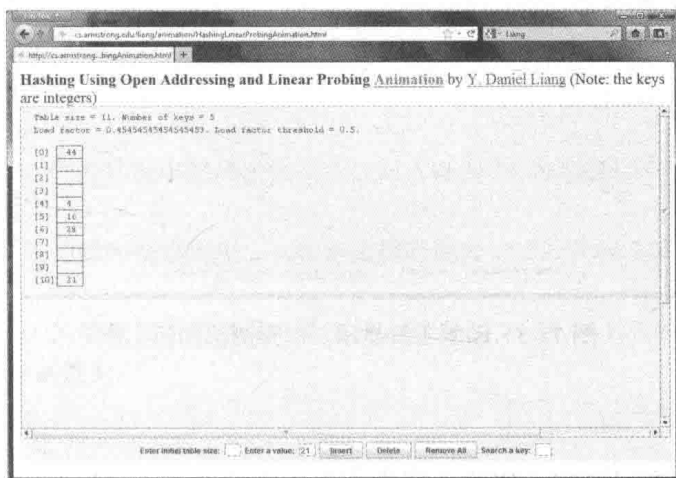


图 27-3 动画工具显示线性探测法如何工作

27.4.2 二次探测法

二次探测法(quadratic probing)可以避免线性探测法产生的成簇的问题。线性探测法从索引 k 位置审查连续单元。二次探测法则从索引为 $(k+j^2) \% N$ 位置的单元开始审查,其中 $j \geq 0$ 。即 $k \% N$, $(k+1) \% N$, $(k+4) \% N$, $(k+9) \% N$, 以此类推,如图 27-4 所示。

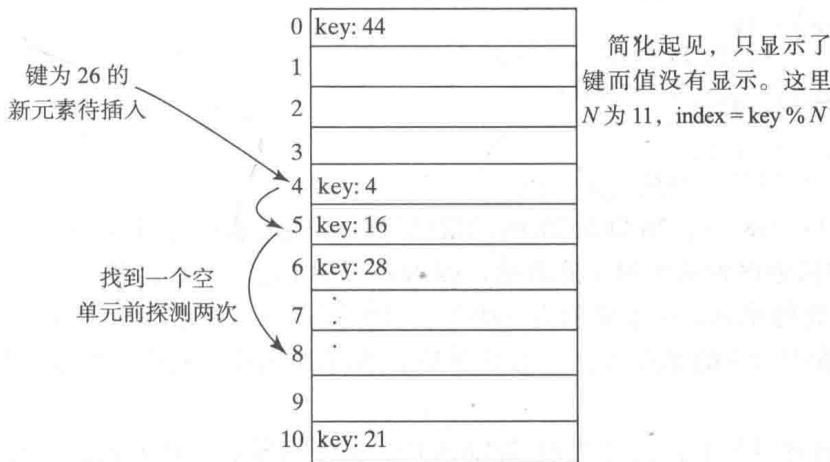


图 27-4 二次探测法以 j^2 ($j=1,2,3,\dots$) 递增下一个索引

除了搜索序列的修改外,二次探测法和线性探测法具有同样的工作机制。二次探测法避免了线性探测法的成簇问题,但是有自己本身的成簇问题,称为二次成簇 (secondary clustering); 即在一个被占据的条目处产生冲突的条目将采用同样的探测序列。

线性探测法可以保证只要表不是满的,一个可用的单元总是可以被找到用于插入新的元素。然而,二次探测法不能保证这个。

{ } 教学注意: 参见网址 www.cs.armstrong.edu/liang/animation/HashingQuadraticProbingAnimation.html, 获取二次探测法如何工作的交互式 GUI 演示, 如图 27-5 所示。

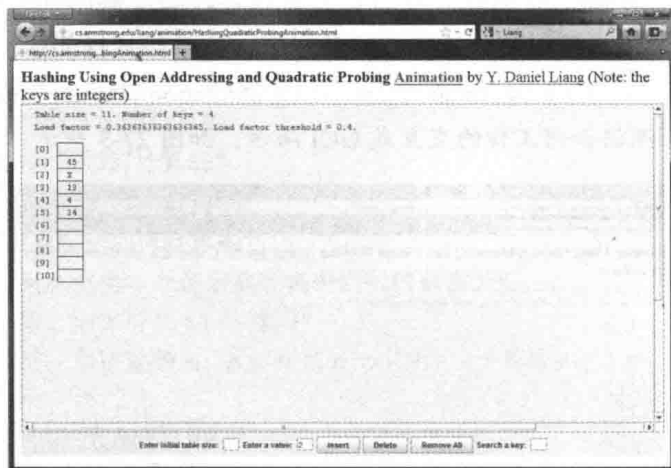


图 27-5 动画工具显示二次探测法如何工作

27.4.3 再哈希法

另外一个避免成簇问题的开放地址模式称为再哈希法 (double hashing)。从初始索引 k 开始, 线性探测法和二次探测法都对 k 增加一个值来定义一个搜索序列。对于线性探测法来说增量为 1, 对于二次探测法来说增量为 j^2 。这些增量都独立于键。再哈希法在键上应用一个二次散列函数 $h'(key)$ 来确定增量, 从而避免成簇问题。具体来说, 再哈希法审查索引为 $(k+j*h'(key)) \% N$ 处的单元, 其中 $j \geq 0$, 即 $k \% N$, $(k+h'(key)) \% N$, $(k+2*h'(key)) \% N$, $(k+3*h'(key)) \% N$, 以此类推。

例如, 让一个大小为 11 的散列表的主散列函数 h 和二次散列函数 h' 如下定义:

$h(key) = key \% 11$;
 $h'(key) = 7 - key \% 7$;

对于搜索键 12, 则有

$h(12) = 12 \% 11 = 1$;
 $h'(12) = 7 - 12 \% 7 = 2$;

假设键为 45、58、4、28 以及 21 的元素已经位于散列表中, 现在插入键为 12 的元素。对于键为 12 的探索序列从索引 1 处开始。因为索引为 1 的单元已经被占据, 搜索下一个索引为 $3(1+1*2)$ 处的单元。由于索引为 3 的单元已经被占据, 搜索下一个索引为 $5(1+2*2)$ 处的单元。由于索引为 5 的单元为空, 则键为 12 的元素插入该单元中。搜索过程在图 27-6 中展示。

探索序列的索引如下: 1,3,5,7,9,0,2,4,6,8,10。该序列覆盖了整个表。应该设计函数来产生一个覆盖整个表的探索序列。注意, 二次函数不能具有一个为 0 的值, 因为 0 不是增量。

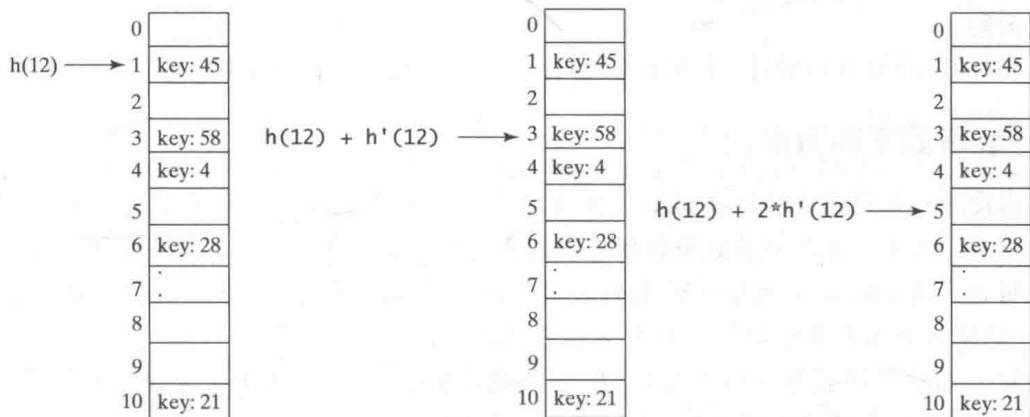


图 27-6 再哈希法中的二次散列函数确定探寻序列中的下一个索引的增量

复习题

- 27.10 什么是开放地址法？什么是线性探测法？什么是二次探测法？什么是再哈希法？
- 27.11 描述线性探测产生的成簇问题。
- 27.12 什么是二次成簇？
- 27.13 显示在大小为 11 的散列表中，使用线性探测法插入键为 34, 29, 53, 44, 120, 39, 45 以及 40 的条目的情形。
- 27.14 显示在大小为 11 的散列表中，使用二次探测法插入键为 34, 29, 53, 44, 120, 39, 45 以及 40 的条目的情形。
- 27.15 显示在大小为 11 的散列表中，使用再哈希法插入键为 34, 29, 53, 44, 120, 39, 45 以及 40 的条目的情形，其中再哈希函数为：

$$h(k) = k \% 11;$$

$$h'(k) = 7 - k \% 7;$$

27.5 使用链地址法处理冲突

要点提示：链地址法将具有同样的散列索引的条目都放在一个位置，而不是寻找一个新的位置。链地址法的每个位置使用一个桶来放置多个条目。

可以使用数组，ArrayList 或者 LinkedList 来实现一个桶。这里将使用 LinkedList 来作为演示。可以将散列表的每个单元视为指向一个链表头的引用，而链表中的元素从头部链接在一起，如图 27-7 所示。

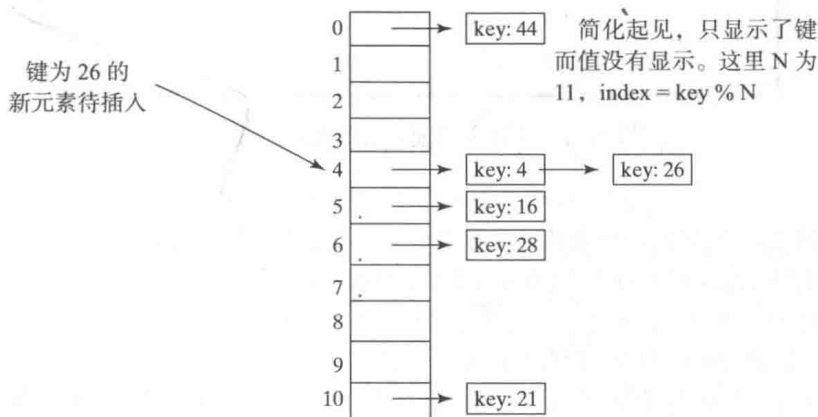


图 27-7 链地址法将具有同样的散列索引的条目放在一个桶内

复习题

27.16 显示在大小为 11 的散列表中使链地址法插入键为 34,29,53,44,120,39,45 以及 40 的条目的情形。

27.6 装填因子和再散列

要点提示：装填因子 (load factor) 衡量一个散列表有多满。如果装填因子溢出，则增加散列表的大小，并重新装载条目到一个新的更大的散列表中。这称为再散列。

装填因子 λ (lamda) 衡量一个散列表有多满。它是元素数目和散列表大小的比例，即 $\lambda = n/N$ ，这里 n 表示元素的数目，而 N 表示散列表中位置的数目。

注意，如果散列表为空则 λ 为 0。对于开放地址法， λ 介于 0 和 1 之间。如果散列表满了，则 λ 为 1。对于链地址法而言， λ 可能为任意值。

当 λ 增加时，冲突的可能性增大。研究表明，对于开放地址法而言，需要维持装填因子在 0.5 以下，而对于链地址法而言，维持在 0.9 以下。

将装填因子保持在一定的阈值下对于散列的性能是非常重要的。Java API 中 `java.util.HashMap` 类的实现中，采用了阈值 0.75。一旦装填因子超过阈值，则需要增加散列表的大小，并将映射表中所有条目再散列 (rehash) 到一个更大的散列表中。注意需要修改散列函数，因为散列表的大小被改变了。由于再散列代价比较大，为了减少出现再散列的可能性，应该至少将散列表的大小翻倍。即使需要周期性的再散列，对于映射表来说散列依然是一种高效的实现。

教学注意：参见网址 www.cs.armstrong.edu/liang/animation/HashingUsingSeparateChainingAnimation.html，获取链地址法如何工作的交互式 GUI 演示，如图 27-8 所示。

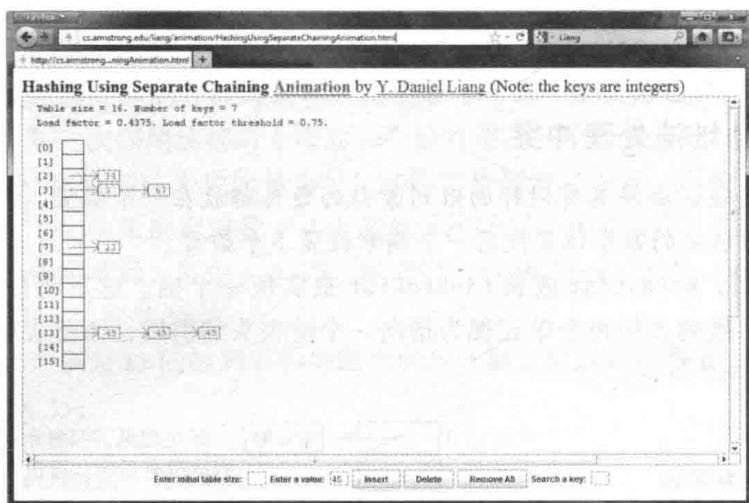


图 27-8 动画工具显示链地址法如何工作

复习题

27.17 什么是装填因子？假设散列表具有初始大小 4，它的装填因子为 0.5；显示在使用线性探测法插入键为 34,29,53,44,120,39,45 以及 40 的条目的散列表。

27.18 假设散列表具有初始大小 4，它的装填因子为 0.5；显示在使用二次探测法插入键为 34,29,53,44,120,39,45 以及 40 的条目的散列表。

27.19 假设散列表具有初始大小 4，它的装填因子为 0.5；显示在使用链地址法插入键为 34,29,53,44,120,39,45 以及 40 的条目的散列表。

27.7 使用散列实现映射表

要点提示：可以使用散列来实现映射表。

现在你理解了散列的概念，了解了如何设计一个好的散列函数，从而将一个键映射到散列表的索引上；了解了如何使用装填因子衡量性能，以及如何通过增加表的大小和再散列来保持性能。本节演示如何使用链地址法来实现映射表。

这里对照 `java.util.Map` 来设计我们自定义的 `Map` 接口，将接口命名为 `MyMap`，具体类命名为 `MyHashMap`，如图 27-9 所示。

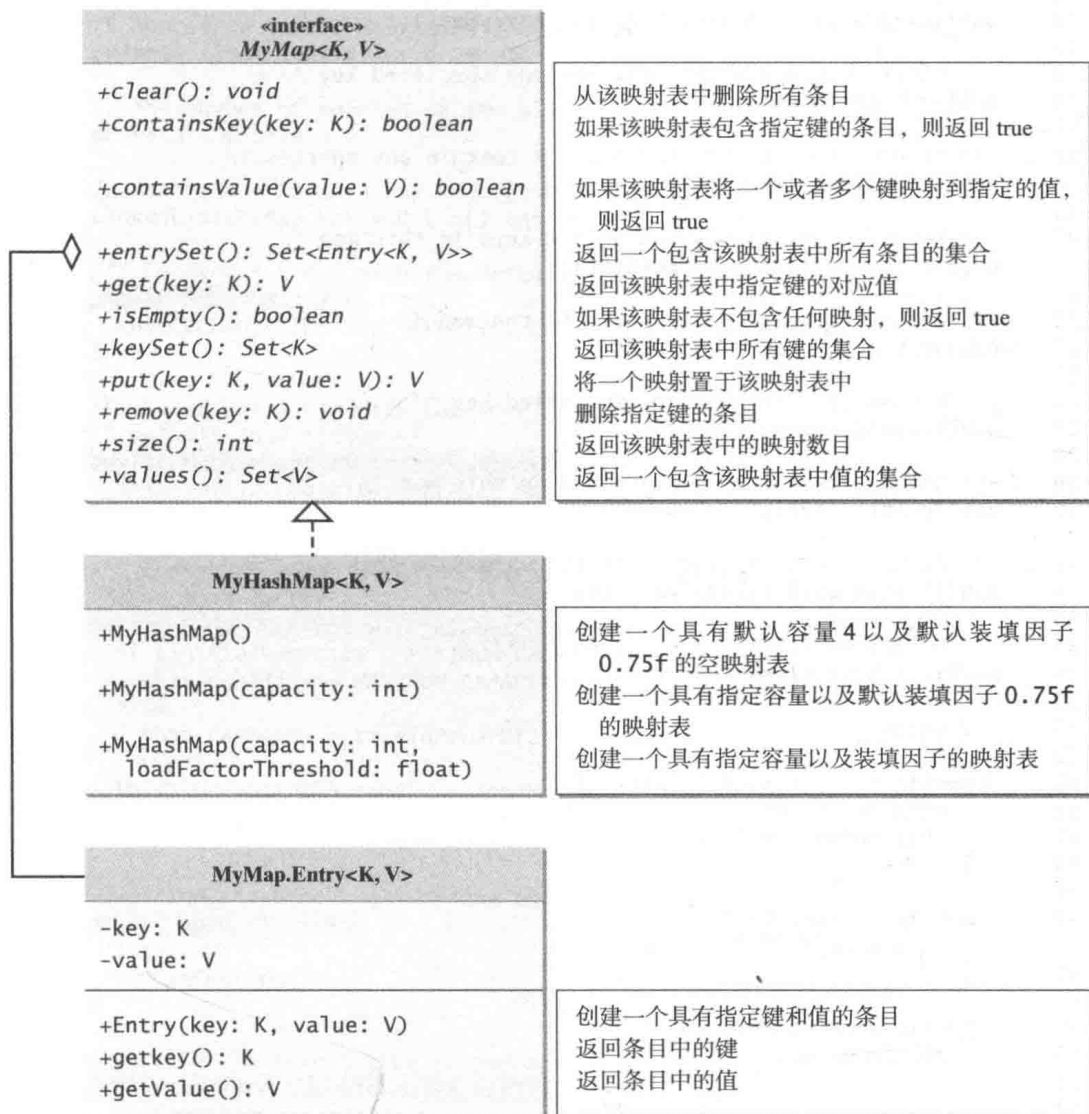


图 27-9 MyHashMap 实现 MyMap 接口

如何实现 `MyHashMap` 呢？如果使用一个 `ArrayList` 并将新的条目存储在列表的最后，搜索时间为 $O(n)$ 。如果使用一棵二叉树实现 `MyHashMap`，在树为良好平衡的情况下搜索时间为 $O(\log n)$ 。然而，可以采用散列来实现 `MyHashMap`，从而获得 $O(1)$ 时间的搜索算法。程序清单 27-1 给出了 `MyMap` 接口，程序清单 27-2 采用链地址法实现 `MyHashMap`。

程序清单 27-1 MyMap.java

```

1  public interface MyMap<K, V> {
2      /** Remove all of the entries from this map */
3      public void clear();
4
5      /** Return true if the specified key is in the map */
6      public boolean containsKey(K key);
7
8      /** Return true if this map contains the specified value */
9      public boolean containsValue(V value);
10
11     /** Return a set of entries in the map */
12     public java.util.Set<Entry<K, V>> entrySet();
13
14     /** Return the value that matches the specified key */
15     public V get(K key);
16
17     /** Return true if this map doesn't contain any entries */
18     public boolean isEmpty();
19
20     /** Return a set consisting of the keys in this map */
21     public java.util.Set<K> keySet();
22
23     /** Add an entry (key, value) into the map */
24     public V put(K key, V value);
25
26     /** Remove an entry for the specified key */
27     public void remove(K key);
28
29     /** Return the number of mappings in this map */
30     public int size();
31
32     /** Return a set consisting of the values in this map */
33     public java.util.Set<V> values();
34
35     /** Define an inner class for Entry */
36     public static class Entry<K, V> {
37         K key;
38         V value;
39
40         public Entry(K key, V value) {
41             this.key = key;
42             this.value = value;
43         }
44
45         public K getKey() {
46             return key;
47         }
48
49         public V getValue() {
50             return value;
51         }
52
53         @Override
54         public String toString() {
55             return "[" + key + ", " + value + "]";
56         }
57     }
58 }

```

程序清单 27-2 MyHashMap.java

```

1  import java.util.LinkedList;
2

```

```

3 public class MyHashMap<K, V> implements MyMap<K, V> {
4     // Define the default hash-table size. Must be a power of 2
5     private static int DEFAULT_INITIAL_CAPACITY = 4;
6
7     // Define the maximum hash-table size.  $1 \ll 30$  is same as  $2^{30}$ 
8     private static int MAXIMUM_CAPACITY = 1 << 30;
9
10    // Current hash-table capacity. Capacity is a power of 2
11    private int capacity;
12
13    // Define default load factor
14    private static float DEFAULT_MAX_LOAD_FACTOR = 0.75f;
15
16    // Specify a load factor used in the hash table
17    private float loadFactorThreshold;
18
19    // The number of entries in the map
20    private int size = 0;
21
22    // Hash table is an array with each cell being a linked list
23    LinkedList<MyMap.Entry<K,V>>[] table;
24
25    /** Construct a map with the default capacity and load factor */
26    public MyHashMap() {
27        this(DEFAULT_INITIAL_CAPACITY, DEFAULT_MAX_LOAD_FACTOR);
28    }
29
30    /** Construct a map with the specified initial capacity and
31     * default load factor */
32    public MyHashMap(int initialCapacity) {
33        this(initialCapacity, DEFAULT_MAX_LOAD_FACTOR);
34    }
35
36    /** Construct a map with the specified initial capacity
37     * and load factor */
38    public MyHashMap(int initialCapacity, float loadFactorThreshold) {
39        if (initialCapacity > MAXIMUM_CAPACITY)
40            this.capacity = MAXIMUM_CAPACITY;
41        else
42            this.capacity = trimToPowerOf2(initialCapacity);
43
44        this.loadFactorThreshold = loadFactorThreshold;
45        table = new LinkedList[capacity];
46    }
47
48    @Override /** Remove all of the entries from this map */
49    public void clear() {
50        size = 0;
51        removeEntries();
52    }
53
54    @Override /** Return true if the specified key is in the map */
55    public boolean containsKey(K key) {
56        if (get(key) != null)
57            return true;
58        else
59            return false;
60    }
61
62    @Override /** Return true if this map contains the value */
63    public boolean containsValue(V value) {
64        for (int i = 0; i < capacity; i++) {
65            if (table[i] != null) {
66                LinkedList<Entry<K, V>> bucket = table[i];
67                for (Entry<K, V> entry: bucket)

```



```

68         if (entry.getValue().equals(value))
69             return true;
70     }
71 }
72
73     return false;
74 }
75
76 @Override /** Return a set of entries in the map */
77 public java.util.Set<MyMap.Entry<K,V>> entrySet() {
78     java.util.Set<MyMap.Entry<K, V>> set =
79         new java.util.HashSet<>();
80
81     for (int i = 0; i < capacity; i++) {
82         if (table[i] != null) {
83             LinkedList<Entry<K, V>> bucket = table[i];
84             for (Entry<K, V> entry: bucket)
85                 set.add(entry);
86         }
87     }
88
89     return set;
90 }
91
92 @Override /** Return the value that matches the specified key */
93 public V get(K key) {
94     int bucketIndex = hash(key.hashCode());
95     if (table[bucketIndex] != null) {
96         LinkedList<Entry<K, V>> bucket = table[bucketIndex];
97         for (Entry<K, V> entry: bucket)
98             if (entry.getKey().equals(key))
99                 return entry.getValue();
100     }
101
102     return null;
103 }
104
105 @Override /** Return true if this map contains no entries */
106 public boolean isEmpty() {
107     return size == 0;
108 }
109
110 @Override /** Return a set consisting of the keys in this map */
111 public java.util.Set<K> keySet() {
112     java.util.Set<K> set = new java.util.HashSet<K>();
113
114     for (int i = 0; i < capacity; i++) {
115         if (table[i] != null) {
116             LinkedList<Entry<K, V>> bucket = table[i];
117             for (Entry<K, V> entry: bucket)
118                 set.add(entry.getKey());
119         }
120     }
121
122     return set;
123 }
124
125 @Override /** Add an entry (key, value) into the map */
126 public V put(K key, V value) {
127     if (get(key) != null) { // The key is already in the map
128         int bucketIndex = hash(key.hashCode());
129         LinkedList<Entry<K, V>> bucket = table[bucketIndex];
130         for (Entry<K, V> entry: bucket)
131             if (entry.getKey().equals(key)) {
132                 V oldValue = entry.getValue();
133                 // Replace old value with new value

```

```

134         entry.value = value;
135         // Return the old value for the key
136         return oldValue;
137     }
138 }
139
140 // Check load factor
141 if (size >= capacity * loadFactorThreshold) {
142     if (capacity == MAXIMUM_CAPACITY)
143         throw new RuntimeException("Exceeding maximum capacity");
144     rehash();
145 }
146
147
148 int bucketIndex = hash(key.hashCode());
149
150 // Create a linked list for the bucket if not already created
151 if (table[bucketIndex] == null) {
152     table[bucketIndex] = new LinkedList<Entry<K, V>>();
153 }
154
155 // Add a new entry (key, value) to hashTable[index]
156 table[bucketIndex].add(new MyMap.Entry<K, V>(key, value));
157
158 size++; // Increase size
159
160 return value;
161 }
162
163 @Override /** Remove the entries for the specified key */
164 public void remove(K key) {
165     int bucketIndex = hash(key.hashCode());
166
167     // Remove the first entry that matches the key from a bucket
168     if (table[bucketIndex] != null) {
169         LinkedList<Entry<K, V>> bucket = table[bucketIndex];
170         for (Entry<K, V> entry: bucket)
171             if (entry.getKey().equals(key)) {
172                 bucket.remove(entry);
173                 size--; // Decrease size
174                 break; // Remove just one entry that matches the key
175             }
176     }
177 }
178
179 @Override /** Return the number of entries in this map */
180 public int size() {
181     return size;
182 }
183
184 @Override /** Return a set consisting of the values in this map */
185 public java.util.Set<V> values() {
186     java.util.Set<V> set = new java.util.HashSet<>();
187
188     for (int i = 0; i < capacity; i++) {
189         if (table[i] != null) {
190             LinkedList<Entry<K, V>> bucket = table[i];
191             for (Entry<K, V> entry: bucket)
192                 set.add(entry.getValue());
193         }
194     }
195
196     return set;
197 }
198
199 /** Hash function */

```

```

200 private int hash(int hashCode) {
201     return supplementalHash(hashCode) & (capacity - 1);
202 }
203
204 /** Ensure the hashing is evenly distributed */
205 private static int supplementalHash(int h) {
206     h ^= (h >>> 20) ^ (h >>> 12);
207     return h ^ (h >>> 7) ^ (h >>> 4);
208 }
209
210 /** Return a power of 2 for initialCapacity */
211 private int trimToPowerOf2(int initialCapacity) {
212     int capacity = 1;
213     while (capacity < initialCapacity) {
214         capacity <=> 1; // Same as capacity *= 2. <= is more efficient
215     }
216
217     return capacity;
218 }
219
220 /** Remove all entries from each bucket */
221 private void removeEntries() {
222     for (int i = 0; i < capacity; i++) {
223         if (table[i] != null) {
224             table[i].clear();
225         }
226     }
227 }
228
229 /** Rehash the map */
230 private void rehash() {
231     java.util.Set<Entry<K, V>> set = entrySet(); // Get entries
232     capacity <=> 1; // Same as capacity *= 2. <= is more efficient
233     table = new LinkedList[capacity]; // Create a new hash table
234     size = 0; // Reset size to 0
235
236     for (Entry<K, V> entry: set) {
237         put(entry.getKey(), entry.getValue()); // Store to new table
238     }
239 }
240
241 @Override /** Return a string representation for this map */
242 public String toString() {
243     StringBuilder builder = new StringBuilder("");
244
245     for (int i = 0; i < capacity; i++) {
246         if (table[i] != null && table[i].size() > 0)
247             for (Entry<K, V> entry: table[i])
248                 builder.append(entry);
249     }
250
251     builder.append("");
252     return builder.toString();
253 }
254 }

```

MyHashMap 类采用链地址法实现 MyMap 接口。确定散列表大小和装填因子的参数在类中定义。默认的初始容量为 4 (第 5 行), 最大容量为 2^{30} (第 8 行)。当前散列表容量设计为一个 2 的幂的值 (第 11 行)。默认的装填因子阈值为 0.75f (第 14 行)。可以在构建一个映射表的时候给出一个装填因子阈值。自定义的装填因子阈值保存在 loadFactorThreshold 中 (第 17 行)。数据域 size 表示映射表中的条目数 (第 20 行)。散列表是一个数组, 数组中的每个单元是一个链表 (第 23 行)。

提供了三个构造方法来构建一个映射表。可以使用无参构造方法来构建具有默认的容量和装填因子阈值的映射表（第 26 ~ 28 行），可以构造具有指定的容量和默认的装填因子阈值的映射表（第 32 ~ 34 行），以及构建具有指定的容量和装填因子阈值的映射表（第 38 ~ 46 行）。

`clear` 方法从映射表中删除所有的条目（第 49 ~ 52 行）。该方法调用 `removeEntries()`，这将删除桶中的所有条目（第 221 ~ 227 行）。`removeEntries()` 方法用 $O(capacity)$ 的时间来清除表中的所有条目。

`containsKey(key)` 方法通过调用 `get` 方法（第 55 ~ 60 行）检测指定的键是否在映射表中。由于 `get` 方法耗费 $O(1)$ 时间，`containsKey(key)` 方法也耗费 $O(1)$ 时间。

`containsValue(value)` 方法检测某值是否存在于映射表中（第 63 ~ 74 行）。该方法耗费 $O(capacity + size)$ 时间。实际上是 $O(capacity)$ ，因为 $capacity > size$ 。

`entrySet()` 方法返回一个包含映射表中所有条目的集合（第 77 ~ 90 行）。该方法需要 $O(capacity)$ 时间。

`get(key)` 方法返回具有指定键的第一个条目的值（第 93 ~ 103 行）。该方法需要 $O(1)$ 时间。

`isEmpty()` 方法在映射表为空的情况下简单地返回 `true`（第 106 ~ 108 行）。该方法需要 $O(1)$ 时间。

`keySet()` 方法返回一个包含映射表中所有键的集合。该方法从每个桶中寻找键并将它们加入一个集合中（第 111 ~ 123 行）。该方法需要 $O(capacity)$ 时间。

`put(key, value)` 方法添加一个新的条目到映射表中。该方法首先测试该键是否已经在映射表中（第 127 行），如果是，它定位该条目，并将该键所在的条目的旧值替换成新值（第 134 行），并返回旧值（第 136 行）。如果键不在映射表中，则在映射表中产生一个新的条目（第 156 行）。插入新的条目之前，该方法检测大小是否超过了装填因子的阈值（第 141 行）。如果是，程序调用 `rehash()`（第 145 行）来增加容量，并将条目保存到新的更大的散列表中。

`rehash()` 方法首先复制所有集合中的条目（第 231 行），将容量翻倍（第 232 行），创建一个新的散列表（第 233 行），并将大小重置为 0（第 234 行）。然后该方法将所有的条目复制到一个新的散列表中（第 236 ~ 238 行）。`Rehash` 方法花费 $O(capacity)$ 时间。如果不执行再散列，`put` 方法花费 $O(1)$ 时间来添加一个新的条目。

`remove(key)` 方法删除映射表中指定键的条目（第 164 ~ 177 行）。该方法花费 $O(1)$ 时间。

`size()` 方法简单地返回映射表的大小（第 180 ~ 182 行）。该方法花费 $O(1)$ 时间。

`value()` 方法返回映射表中所有的值。该方法从所有的桶中检测每个条目，然后添加值到一个集合中（第 185 ~ 197 行）。该方法花费 $O(capacity)$ 时间。

`hash()` 方法调用 `supplementalHash` 方法来确保为散列表生成索引的散列是均匀分布的（第 200 ~ 208 行）。该方法花费 $O(1)$ 时间。

表 27-1 总结了 `MyHashMap` 中方法的时间复杂性。

表 27-1 `MyHashMap` 中方法的时间复杂性

方法	时间	方法	时间
<code>clear()</code>	$O(capacity)$	<code>keySet()</code>	$O(capacity)$
<code>containsKey(key: Key)</code>	$O(1)$	<code>put(key: K, value: V)</code>	$O(1)$
<code>containsValue(value: V)</code>	$O(capacity)$	<code>remove(key: K)</code>	$O(1)$
<code>entrySet()</code>	$O(capacity)$	<code>size()</code>	$O(1)$
<code>get(key: K)</code>	$O(1)$	<code>values()</code>	$O(capacity)$
<code>isEmpty()</code>	$O(1)$	<code>rehash()</code>	$O(capacity)$

由于再散列并不经常发生, `put` 方法的时间复杂性为 $O(1)$ 。注意, `clear`、`entrySet`、`keySet`、`values` 以及 `rehash` 方法的复杂性依赖于 `capacity`, 因此应该精心选择初始容量来避免这些方法的较差性能。

程序清单 27-3 给出了应用 `MyHashMap` 的测试程序。

程序清单 27-3 TestMyHashMap.java

```

1 public class TestMyHashMap {
2     public static void main(String[] args) {
3         // Create a map
4         MyMap<String, Integer> map = new MyHashMap<>();
5         map.put("Smith", 30);
6         map.put("Anderson", 31);
7         map.put("Lewis", 29);
8         map.put("Cook", 29);
9         map.put("Smith", 65);
10
11        System.out.println("Entries in map: " + map);
12
13        System.out.println("The age for Lewis is " +
14            map.get("Lewis"));
15
16        System.out.println("Is Smith in the map? " +
17            map.containsKey("Smith"));
18        System.out.println("Is age 33 in the map? " +
19            map.containsValue(33));
20
21        map.remove("Smith");
22        System.out.println("Entries in map: " + map);
23
24        map.clear();
25        System.out.println("Entries in map: " + map);
26    }
27 }

```

```

Entries in map: [[Anderson, 31][Smith, 65][Lewis, 29][Cook, 29]]
The age for Lewis is 29
Is Smith in the map? true
Is age 33 in the map? false
Entries in map: [[Anderson, 31][Lewis, 29][Cook, 29]]
Entries in map: []

```

该程序应用 `MyHashMap` 创建一个映射表 (第 4 行), 并添加 5 个条目到映射表中 (第 5 ~ 9 行)。第 5 行添加键 `Smith` 和相应的值 30, 第 9 行添加键 `Smith` 和相应的值 65。后者的值替换了前者的值。映射表实际上只有 4 个条目。程序显示了映射表中的条目 (第 11 行), 对于一个键得到相应的值 (第 14 行), 检测映射表是否包含某个键 (第 17 行) 以及某个值 (第 19 行), 删除键 `Smith` 的条目 (第 21 行), 然后重新显示映射表中的条目 (第 22 行)。最后, 程序清除映射表 (第 24 行) 并显示一个空的映射表 (第 25 行)。

✓ 复习题

- 27.20 程序清单 27-2 中, 第 8 行的 `1 << 30` 是什么? `1 << 1`, `1 << 2` 以及 `1 << 3` 的结果是什么整数?
- 27.21 `32 >> 1`, `32 >> 2`, `32 >> 3` 以及 `32 >> 4` 的结果是什么整数?
- 27.22 程序清单 27-2 中, 如果 `LinkedList` 替换为 `ArrayList`, 程序还能工作吗? 程序清单 27-2 中, 如何将第 55 ~ 59 行的代码替换为一行代码?
- 27.23 描述 `MyHashMap` 类中 `put(key, value)` 方法是如何实现的?
- 27.24 程序清单 27-5 中, `supplementalHash` 方法声明为静态的, `hash` 方法可以声明为静态的吗?

27.25 给出下面代码的输出结果。

```

MyMap<String, String> map = new MyHashMap<>();
map.put("Texas", "Dallas");
map.put("Oklahoma", "Norman");
map.put("Texas", "Austin");
map.put("Oklahoma", "Tulsa");

System.out.println(map.get("Texas"));
System.out.println(map.size());

```

27.8 使用散列实现集合

要点提示：可以使用散列映射表来实现散列集。

集合（第 21 章中介绍过）是一种存储不同值的数据结构。Java 合集框架定义了 `java.util.Set` 接口来对集合建模。三种具体的实现是 `java.util.HashSet`、`java.util.LinkedHashSet` 以及 `java.util.TreeSet`。`java.util.HashSet` 采用散列实现，`java.util.LinkedHashSet` 采用 `LinkedList` 实现，`java.util.TreeSet` 采用红黑树实现。

可以采用实现 `MyHashMap` 同样的方式来实现 `MyHashSet`。唯一的不同之处在于键 / 值对存储在映射表中，而元素存储在集合中。

我们对着 `java.util.Set` 来设计自定义的 `Set` 接口，将接口命名为 `MySet`，设计具体类 `MyHashSet`，如图 27-10 所示。

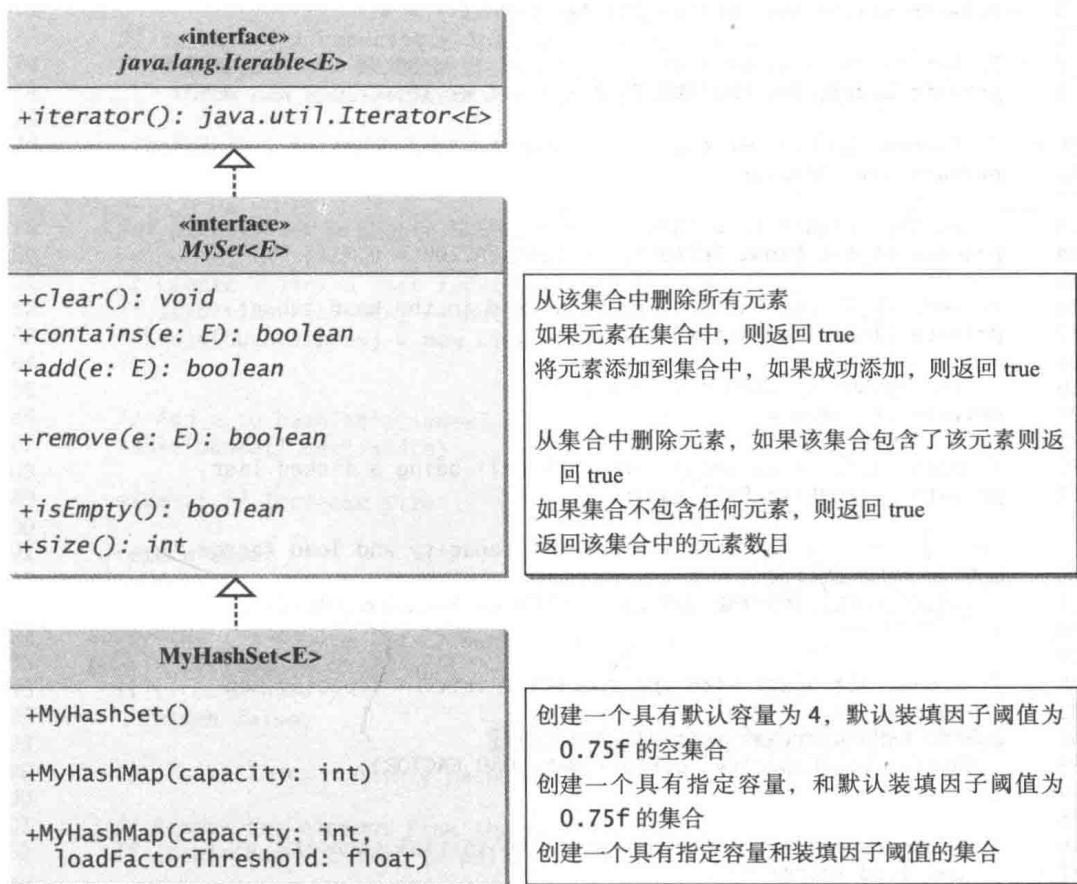


图 27-10 `MyHashSet` 实现 `MySet` 接口

程序清单 27-4 给出了 `MySet` 接口，程序清单 27-5 采用链地址法实现了 `MyHashSet`。

程序清单 27-4 MySet.java

```

1  public interface MySet<E> extends java.lang.Iterable<E> {
2      /** Remove all elements from this set */
3      public void clear();
4
5      /** Return true if the element is in the set */
6      public boolean contains(E e);
7
8      /** Add an element to the set */
9      public boolean add(E e);
10
11     /** Remove the element from the set */
12     public boolean remove(E e);
13
14     /** Return true if the set doesn't contain any elements */
15     public boolean isEmpty();
16
17     /** Return the number of elements in the set */
18     public int size();
19 }

```

程序清单 27-5 MyHashSet.java

```

1  import java.util.LinkedList;
2
3  public class MyHashSet<E> implements MySet<E> {
4      // Define the default hash-table size. Must be a power of 2
5      private static int DEFAULT_INITIAL_CAPACITY = 4;
6
7      // Define the maximum hash-table size. 1 << 30 is same as 2^30
8      private static int MAXIMUM_CAPACITY = 1 << 30;
9
10     // Current hash-table capacity. Capacity is a power of 2
11     private int capacity;
12
13     // Define default load factor
14     private static float DEFAULT_MAX_LOAD_FACTOR = 0.75f;
15
16     // Specify a load-factor threshold used in the hash table
17     private float loadFactorThreshold;
18
19     // The number of elements in the set
20     private int size = 0;
21
22     // Hash table is an array with each cell being a linked list
23     private LinkedList<E>[] table;
24
25     /** Construct a set with the default capacity and load factor */
26     public MyHashSet() {
27         this(DEFAULT_INITIAL_CAPACITY, DEFAULT_MAX_LOAD_FACTOR);
28     }
29
30     /** Construct a set with the specified initial capacity and
31      * default load factor */
32     public MyHashSet(int initialCapacity) {
33         this(initialCapacity, DEFAULT_MAX_LOAD_FACTOR);
34     }
35
36     /** Construct a set with the specified initial capacity
37      * and load factor */
38     public MyHashSet(int initialCapacity, float loadFactorThreshold) {
39         if (initialCapacity > MAXIMUM_CAPACITY)
40             this.capacity = MAXIMUM_CAPACITY;
41         else
42             this.capacity = trimToPowerOf2(initialCapacity);

```



```

43
44     this.loadFactorThreshold = loadFactorThreshold;
45     table = new LinkedList[capacity];
46 }
47
48 @Override /** Remove all elements from this set */
49 public void clear() {
50     size = 0;
51     removeElements();
52 }
53
54 @Override /** Return true if the element is in the set */
55 public boolean contains(E e) {
56     int bucketIndex = hash(e.hashCode());
57     if (table[bucketIndex] != null) {
58         LinkedList<E> bucket = table[bucketIndex];
59         for (E element: bucket)
60             if (element.equals(e))
61                 return true;
62     }
63
64     return false;
65 }
66
67 @Override /** Add an element to the set */
68 public boolean add(E e) {
69     if (contains(e)) // Duplicate element not stored
70         return false;
71
72     if (size + 1 > capacity * loadFactorThreshold) {
73         if (capacity == MAXIMUM_CAPACITY)
74             throw new RuntimeException("Exceeding maximum capacity");
75
76         rehash();
77     }
78
79     int bucketIndex = hash(e.hashCode());
80
81     // Create a linked list for the bucket if not already created
82     if (table[bucketIndex] == null) {
83         table[bucketIndex] = new LinkedList<E>();
84     }
85
86     // Add e to hashTable[index]
87     table[bucketIndex].add(e);
88
89     size++; // Increase size
90
91     return true;
92 }
93
94 @Override /** Remove the element from the set */
95 public boolean remove(E e) {
96     if (!contains(e))
97         return false;
98
99     int bucketIndex = hash(e.hashCode());
100
101     // Remove the element from the bucket
102     if (table[bucketIndex] != null) {
103         LinkedList<E> bucket = table[bucketIndex];
104         for (E element: bucket)
105             if (e.equals(element)) {
106                 bucket.remove(element);
107                 break;

```

```

108     }
109 }
110
111     size--; // Decrease size
112
113     return true;
114 }
115
116 @Override /** Return true if the set contain no elements */
117 public boolean isEmpty() {
118     return size == 0;
119 }
120
121 @Override /** Return the number of elements in the set */
122 public int size() {
123     return size;
124 }
125
126 @Override /** Return an iterator for the elements in this set */
127 public java.util.Iterator<E> iterator() {
128     return new MyHashSetIterator(this);
129 }
130
131 /** Inner class for iterator */
132 private class MyHashSetIterator implements java.util.Iterator<E> {
133     // Store the elements in a list
134     private java.util.ArrayList<E> list;
135     private int current = 0; // Point to the current element in list
136     private MyHashSet<E> set;
137
138     /** Create a list from the set */
139     public MyHashSetIterator(MyHashSet<E> set) {
140         this.set = set;
141         list = setToList();
142     }
143
144     @Override /** Next element for traversing? */
145     public boolean hasNext() {
146         if (current < list.size())
147             return true;
148
149         return false;
150     }
151
152     @Override /** Get current element and move cursor to the next */
153     public E next() {
154         return list.get(current++);
155     }
156
157     /** Remove the current element and refresh the list */
158     public void remove() {
159         // Delete the current element from the hash set
160         set.remove(list.get(current));
161         list.remove(current); // Remove current element from the list
162     }
163 }
164
165 /** Hash function */
166 private int hash(int hashCode) {
167     return supplementalHash(hashCode) & (capacity - 1);
168 }
169
170 /** Ensure the hashing is evenly distributed */
171 private static int supplementalHash(int h) {
172     h ^= (h >>> 20) ^ (h >>> 12);

```

```

173     return h ^ (h >>> 7) ^ (h >>> 4);
174 }
175
176 /** Return a power of 2 for initialCapacity */
177 private int trimToPowerOf2(int initialCapacity) {
178     int capacity = 1;
179     while (capacity < initialCapacity) {
180         capacity <=<= 1; // Same as capacity *= 2. <=<= is more efficient
181     }
182
183     return capacity;
184 }
185
186 /** Remove all e from each bucket */
187 private void removeElements() {
188     for (int i = 0; i < capacity; i++) {
189         if (table[i] != null) {
190             table[i].clear();
191         }
192     }
193 }
194
195 /** Rehash the set */
196 private void rehash() {
197     java.util.ArrayList<E> list = setToList(); // Copy to a list
198     capacity <=<= 1; // Same as capacity *= 2. <=<= is more efficient
199     table = new LinkedList[capacity]; // Create a new hash table
200     size = 0;
201
202     for (E element: list) {
203         add(element); // Add from the old table to the new table
204     }
205 }
206
207 /** Copy elements in the hash set to an array list */
208 private java.util.ArrayList<E> setToList() {
209     java.util.ArrayList<E> list = new java.util.ArrayList<>();
210
211     for (int i = 0; i < capacity; i++) {
212         if (table[i] != null) {
213             for (E e: table[i]) {
214                 list.add(e);
215             }
216         }
217     }
218
219     return list;
220 }
221
222 @Override /** Return a string representation for this set */
223 public String toString() {
224     java.util.ArrayList<E> list = setToList();
225     StringBuilder builder = new StringBuilder("[");
226
227     // Add the elements except the last one to the string builder
228     for (int i = 0; i < list.size() - 1; i++) {
229         builder.append(list.get(i) + ", ");
230     }
231
232     // Add the last element in the list to the string builder
233     if (list.size() == 0)
234         builder.append("]");
235     else
236         builder.append(list.get(list.size() - 1) + "]");
237 }

```

```
238     return builder.toString();
239 }
240 }
```

MyHashSet 类使用链地址法实现了 MySet 接口。实现 MyHashSet 很类似于实现 MyHashMap，除以下不同之处：

1) 对于 MyHashSet 来说，元素存储在散列表中，而对于 MyHashMap 来说，条目（键 / 值对）存储在散列表中。

2) MySet 继承自 java.lang.Iterable，MyHashSet 实现了 MySet 并重写 iterator()。因此 MyHashSet 中的元素是可遍历的。

提供了三个构造方法来构建一个集合。可以使用无参构造方法来构建具有默认的容量和装填因子阈值的默认集合（第 26 ~ 28 行），可以构造具有指定的容量和默认的装填因子阈值的集合（第 32 ~ 34 行），以及构建具有指定的容量和装填因子阈值的集合（第 38 ~ 46 行）。

clear 方法从集合中删除所有的条目（第 49 ~ 52 行）。该方法调用 removeElements()，这将删除所有表中的单元（第 190 行）。每个表中的单元是一个存储了具有相同散列码的元素的链表。removeElements() 方法花费 $O(capacity)$ 的时间。

contains (element) 方法通过审查指定的桶是否包含元素（第 55 ~ 65 行），来检测指定的键是否在集合中。该方法花费 $O(1)$ 时间。

add(element) 方法添加一个新的元素到集合中。该方法首先检测该元素是否已经在集合中（第 69 行）。如果是，该方法返回 false。接着该方法检测是否大小超出了装填因子的阈值（第 72 行）。如果是，该程序调用 rehash()（第 76 行）来增加容量并将元素存储到新的更大的散列表中。

rehash() 方法首先复制线性表中的所有元素（第 197 行），将容量翻倍（第 198 行），创建一个新的散列表（第 199 行），并将大小重置为 0（第 200 行）。然后该方法将所有元素复制到一个新的更大的散列表中（第 202 ~ 204 行）。rehash 方法花费 $O(capacity)$ 时间。如果不执行再散列，add 方法花费 $O(1)$ 时间来添加一个新的元素。

remove(element) 方法删除集合中指定的元素（第 95 ~ 114 行）。该方法花费 $O(1)$ 时间。

size() 方法简单地返回集合中元素的数目（第 122 ~ 124 行）。该方法花费 $O(1)$ 时间。

iterator() 方法返回一个 java.util.Iterator 的实例。MyHashSetIterator 类实现 java.util.Iterator 来创建一个前向遍历器。当构建一个 MyHashSetIterator 时，复制集合所有的元素到一个线性表中（第 141 行）。变量 current 指向线性表中的元素。初始状态，current 为 0（第 135 行）表示指向线性表中的第一个元素。MyHashSetIterator 实现了 java.util.Iterator 中的方法 hasNext()、next() 以及 remove()。如果 $current < list.size()$ ，则调用 hasNext() 返回 true。调用 next() 返回当前元素并将 current 移动指向下一个元素（第 153 行）。调用 remove() 从集合的遍历器中删除当前元素。

hash() 方法调用 supplementalHash 方法来确保为散列表生成索引的散列是均匀分布的（第 166 ~ 174 行）。该方法花费 $O(1)$ 时间。

表 27-2 结了 MyHashSet 中方法的时间复杂性。

表 27-2 MyHashSet 中方法的时间复杂度

方法	时间
clear()	$O(capacity)$
contains(e: E)	$O(1)$
add(e: E)	$O(1)$
remove(e: E)	$O(1)$
isEmpty()	$O(1)$
size()	$O(1)$
iterator()	$O(capacity)$
rehash()	$O(capacity)$

程序清单 27-6 给出了应用 MyHashSet 的测试程序。

程序清单 27-6 TestMyHashSet.java

```
1 public class TestMyHashSet {
2     public static void main(String[] args) {
3         // Create a MyHashSet
4         MySet<String> set = new MyHashSet<>();
5         set.add("Smith");
6         set.add("Anderson");
7         set.add("Lewis");
8         set.add("Cook");
9         set.add("Smith");
10
11        System.out.println("Elements in set: " + set);
12        System.out.println("Number of elements in set: " + set.size());
13        System.out.println("Is Smith in set? " + set.contains("Smith"));
14
15        set.remove("Smith");
16        System.out.print("Names in set in uppercase are ");
17        for (String s: set)
18            System.out.print(s.toUpperCase() + " ");
19
20        set.clear();
21        System.out.println("\nElements in set: " + set);
22    }
23 }
```

```
Elements in set: [Cook, Anderson, Smith, Lewis]
Number of elements in set: 4
Is Smith in set? true
Names in set in uppercase are COOK ANDERSON LEWIS
Elements in set: []
```

该程序应用 MyHashSet 创建一个集合（第 4 行），并添加 5 个元素到集合中（第 5 ~ 9 行）。第 5 行添加 Smith，第 9 行再次添加 Smith。由于只有不重复的元素可以存储在集合中，Smith 只在集合中出现一次。集合中实际上有 4 个元素。程序显示了元素（第 11 行），得到它的大小（第 12 行），检测集合是否包含某个指定的元素（第 13 行），删除一个元素（第 15 行）。由于集合中的元素是可遍历的，程序使用了 foreach 循环来遍历集合中的所有元素（第 17 ~ 18 行）。最后，程序清除集合（第 20 行）并显示一个空的集合（第 21 行）。

✓ 复习题

- 27.26 为什么可以使用 foreach 循环来遍历集合中的元素？
- 27.27 描述 MyHashSet 类中的 add(e) 方法是如何实现的？
- 27.28 程序清单 27-5 中，遍历器中的 remove 方法从集合中删除当前元素。同时它也从内部线性表中删除当前元素（第 161 行）：

```
list.remove(current); // Remove current element from the list
```

这个是必须的吗？

- 27.29 将程序清单 27-5 中的第 146 ~ 149 行的代码替换为一行语句。

关键术语

associative array (关联数组)

cluster (簇)

dictionary (字典)

linear probing (线性探测)

load factor (装填因子)

open addressing (开放地址法)

double hashing (再哈希)
hash code (散列码)
hash function (散列函数)
hash map (散列映射表)
hash set (散列集合)
hash table (散列表)

perfect hash function (完全散列函数)
polynomial hash code (多项式散列码)
quadratic probing (二次探测法)
rehashing (再散列)
secondary clustering (二次成簇)
separate chaining (链地址法)

本章小结

1. 映射表是存储条目的一种数据结构。每个条目包含两部分：键和值。键也称为搜索键，用于查找相应的值。可以使用散列技术来实现映射表，实现使用 $O(1)$ 的时间复杂度来实现查找、获取、插入以及删除。
2. 集合是一种存储元素的数据结构。可以使用散列技术来实现集合，实现使用 $O(1)$ 的时间复杂度来实现查找、获取、插入以及删除。
3. 散列是一种无须执行搜索，即可从一个键得到的索引获取值的技术。典型的散列函数首先将搜索键转化为一个称为散列码的整数值，然后将散列码压缩为散列表中的一个索引。
4. 当两个键映射到散列表中的同样索引上时，冲突发生。通常有两种方法处理冲突：开放地址法和链地址法。
5. 开放地址法是在发生冲突时，在散列表中找到一个开放位置的过程。开放地址法有几种变体：线性探测、二次探测以及再哈希。
6. 链地址法将具有同样散列索引的条目放到同一个位置中，而不是寻找新的位置。链地址法中每个位置称为一个桶。桶是容纳多个条目的容器。

测试题

回答位于网址 www.cs.armstrong.edu/liang/intro10e/quiz.html 的本章测试题。

编程练习题

- **27.1** (应用开放地址法的线性探测法来实现 MyMap) 应用开放地址法的线性探测法创建一个实现 MyMap 的新的具体类。简单起见，使用 $f(\text{key}) = \text{key} \% \text{size}$ 作为散列函数，这里 size 是散列表的大小。初始的，散列表的大小为 4。当装填因子超过阈值 (0.5) 时，表的大小翻倍。
- **27.2** (应用开放地址法的二次探测法来实现 MyMap) 应用开放地址法的二次探测法创建一个实现 MyMap 的新的具体类。简单起见，使用 $f(\text{key}) = \text{key} \% \text{size}$ 作为散列函数，这里 size 是散列表的大小。初始的，散列表的大小为 4。当装填因子超过阈值 (0.5) 时，表的大小翻倍。
- **27.3** (应用开放地址法的再哈希法来实现 MyMap) 应用开放地址法的再哈希法创建一个实现 MyMap 的新的具体类。简单起见，使用 $f(\text{key}) = \text{key} \% \text{size}$ 作为散列函数，这里 size 是散列表的大小。初始的，散列表的大小为 4。当装填因子超过阈值 (0.5) 时，表的大小翻倍。
- **27.4** (修改 MyHashMap 使得可以有重复的键) 修改 MyHashMap 从而允许条目可以有重复的键。需要修改 `put(key, value)` 的实现。同时，添加一个名为 `getAll(key)` 的新方法，返回一个匹配映射表中键的值的集合。
- **27.5** (使用 MyHashMap 实现 MyHashSet) 使用 MyHashMap 实现 MyHashSet。注意，可以使用 (key, key) 创建条目，而不是使用 (key, value)。
- **27.6** (实现线性探测法的动画) 编写程序，实现线性探测法的动画，如图 27-3 所示。可以在程序中修改散列表的初始大小。假设装填因子阈值为 0.75。
- **27.7** (实现链地址法的动画) 编写程序，实现 MyHashMap 的动画，如图 27-8 所示。可以在程序中修

改散列表的初始大小。假设装填因子阈值为 0.75。

****27.8** (实现二次探测法的动画) 编写程序, 实现二次探测法的动画, 如图 27-5 所示。可以在程序中修改散列表的初始大小。假设装填因子阈值为 0.75。

****27.9** (实现字符串的散列码) 编写一个方法, 使用 27.3.2 节中描述的方法返回字符串的散列码, 其中 *b* 取值 31。方法头如下:

```
public static int hashCodeForString(String s)
```

****27.10** (比较 MyHashSet 和 MyArrayList) 程序清单 24-3 定义了 MyArrayList。编写一个程序, 产生 0 到 999999 之间的 1000000 个随机双精度值, 并存储在一个 MyArrayList 和 MyHashSet 中。然后产生 0 到 1999999 之间的 1000000 个随机双精度值的线性表。对于线性表中的每个数字, 检测是否在数组线性表中以及是否在散列集合中。运行程序, 给出对于数组线性表和散列集合的总体测试时间。

****27.11** (SetToList) 编写以下方法, 从一个集合中返回 ArrayList。


```
public static <E> ArrayList<E> setToList(Set<E> s)
```


图及其应用

教学目标

- 使用图对真实世界问题进行建模并解释哥尼斯堡的七孔桥问题（28.1 节）。
- 描述图中的术语：顶点、边、单图、加权 / 非加权图以及有向 / 无向图（28.2 节）。
- 使用线性表、边数组、边对象、邻接矩阵和邻接线性表来表示顶点和边（28.3 节）。
- 使用 Graph 接口、AbstractGraph 类和 UnweightedGraph 类来对图建模（28.4 节）。
- 图形化显示图（28.5 节）。
- 使用 AbstractGraph.Tree 类来表示对图的遍历（28.6 节）。
- 设计并且实现深度优先搜索（28.7 节）。
- 使用深度优先搜索解决连通圆问题（28.8 节）。
- 设计并且实现广度优先搜索（28.9 节）。
- 使用广度优先搜索解决 9 个硬币反面的问题（28.10 节）。

28.1 引言

 **要点提示：** 真实世界的许多问题可以使用图算法解决。

图对现实世界问题的建模和解决非常有用。例如，可以使用图对找寻两座城市之间最小飞行次数的问题进行建模，其中顶点代表城市，边代表两座相邻城市之间的航班，如图 28-1 所示。找寻两座城市之间最小飞行次数的问题就简化为找寻图中两个顶点之间最短路径的问题。

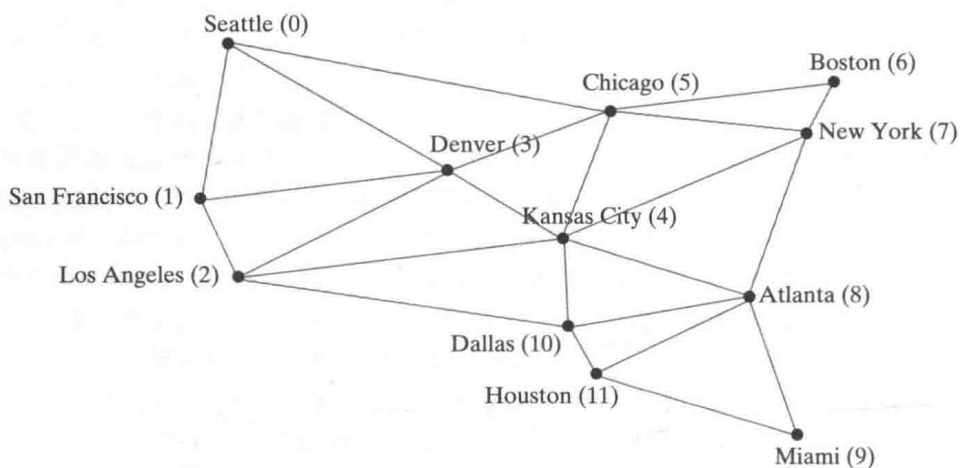


图 28-1 图可以用来对城市之间的飞行次数进行建模

对图的研究也称为图论 (graph theory)。1736 年伦纳德·欧拉创立了图论，当时他将图术语用来解决著名的哥尼斯堡七孔桥问题。位于普鲁士的哥尼斯堡（现俄罗斯的加里宁格勒）被普累格河分开，该河流经两座岛，这座城市和岛由七座桥相连，如图 28-2a 所示。问题在

于, 如何经过每座桥一次且只经过一次, 然后返回起点? 欧拉证明了这是不可能的。

为了证明这个结论, 欧拉首先通过删除所有的街道来提取出哥尼斯堡的地图, 并得到了如图 28-2a 所示的草图。然后, 他将每一块陆地用一个点来替换, 这个点称为顶点 (vertex) 或者结点 (node), 并且将每一座桥用一条线来替换, 这条线称为边 (edge), 如图 28-2b 所示。这种有顶点和边的结构称为图 (graph)。

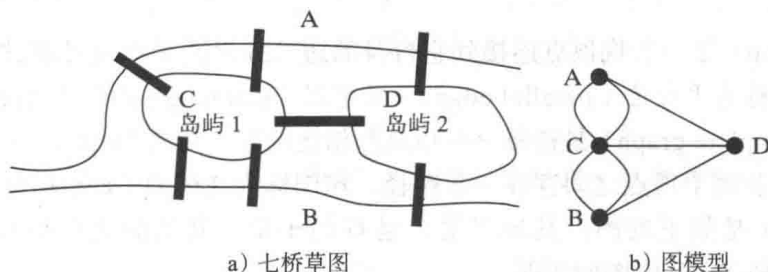


图 28-2 七桥连接岛屿和陆地

当看见图的时候, 我们会询问是否存在一条从任意顶点出发的路径, 这条路径遍历所有的边一次且只有一次, 然后返回起始顶点。欧拉证明了这种路径存在的条件是, 每个顶点必须拥有偶数条边。因此, 哥尼斯堡的七孔桥问题没有解决的方法。

图问题经常通过算法来解决。图算法广泛应用于不同的领域, 例如, 计算机科学、数学、生物学、工程学、经济学、遗传学和社会科学。本章讲述深度优先搜索和广度优先搜索以及它们的应用。下一章将讲述在加权图中找到最小生成树和最短路径的算法, 以及它们的应用。

28.2 基本的图术语

要点提示: 图由顶点以及连接顶点的边所组成。

本章并没有假定读者对图论或者离散数学有任何的预备知识。下面利用简单明了的术语来解释图。

什么是图? 图 (graph) 是一种数学结构, 它表示真实世界中实体之间的关系。例如, 图 28-1 中的图代表了城市间的航班, 图 28-2b 中的图代表了陆地之间的桥梁。

一个图包含了非空的顶点 (结点或者点), 以及一个连接顶点的边的集合。为方便起见, 我们这样定义一个图 $G=(V, E)$, 其中 V 代表顶点的集合, E 代表边的集合。例如, 图 28-1 中图的 V 和 E 分别如下所示:

```
V = {"Seattle", "San Francisco", "Los Angeles",
      "Denver", "Kansas City", "Chicago", "Boston", "New York",
      "Atlanta", "Miami", "Dallas", "Houston"};

E = [{"Seattle", "San Francisco"}, {"Seattle", "Chicago"},
      {"Seattle", "Denver"}, {"San Francisco", "Denver"},
      ...
];
```

图可以是有向的, 也可以是无向的。在有向图 (directed graph) 中, 每条边都有一个方向, 表明可以沿着这条边将一个顶点移动到另一个顶点。可以使用有向图来对父子之间的关系进行建模, 其中从顶点 A 到 B 的边表示 A 是 B 的父结点。图 28-3a 显示了一个有向图。

在无向图 (undirected graph) 中, 可以在顶点之间双向移动它们。图 28-1 中的图是无向的。

边可以是加权的, 也可以是非加权的。例如, 图 28-1 中图的每条边都有一个权值, 表

示两个城市之间的飞行时间。

如果图中的两个顶点被同一条边连接，那么它们被称为相邻的 (adjacent)。相似的，如果两条边连接到同一个顶点，它们也被称为相邻的。在图中，连接两个顶点的边称为连接 (incident) 到这两个顶点。顶点的度 (degree) 就是与这个顶点连接的边的条数。

如果两个顶点是相邻的，那么它们互为邻居 (neighbor)。类似的，两条相邻的边也互为邻居。

一个环 (loop) 是一条将顶点连接到它自身的边。如果两个顶点可通过两条或者多条边相连，这些边就称为平行边 (parallel edge)。简单图 (simple graph) 是指没有环和平行边的图。完全图 (complete graph) 是指每一对顶点都相连的图，如图 28-3b 所示。

如果图中任意两个顶点之间存在一条路径，该图称为连通的 (connected)。一个图 G 的子图 (subgraph) 是如下的图：其顶点集合是 G 的子集，其边的集合是 G 的子集。例如，图 28-3c 中的图是 28-3b 中图的子图。

假设图是连通且无向的。回路 (cycle) 是指始于一个顶点然后终于同一顶点的封闭路径。没有回路的连通图是一棵树 (tree)。图的生成树 (spanning tree) 是一个 G 的连通子图，该子图是包含 G 中所有顶点的树。

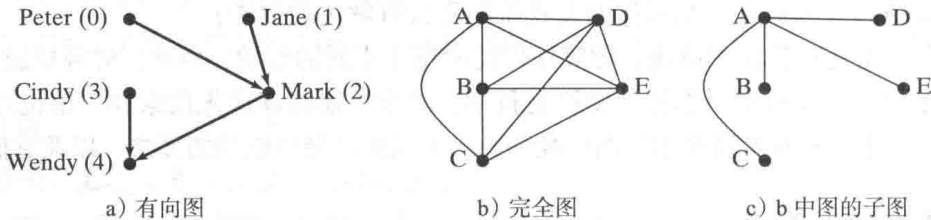


图 28-3 图可以各种形式出现

教学注意：在开始介绍图算法和应用之前，通过网址 www.cs.armstrong.edu/liang/animation/GraphLearningTool.html 提供的交互式工具来了解下图是很有帮助的，如图 28-4 所示。该工具可以让你通过鼠标操作添加 / 删除 / 移动顶点以及绘制边。也可以找到深度优先搜索 (DFS) 树和广度优先搜索 (BFS) 树，以及找到两个顶点之间的最短路径。

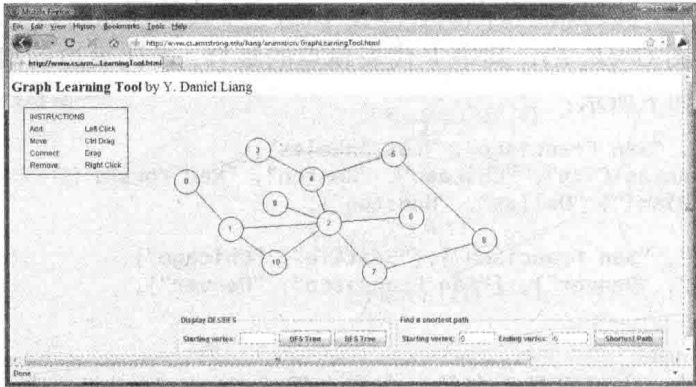


图 28-4 可以使用工具，通过鼠标操作来创建图，以及显示 DFS/BFS 树和最短路径

复习题

- 28.1 什么是著名的哥尼斯堡七桥问题？
- 28.2 什么是图？解释下列术语：无向图、有向图、加权图、顶点的度、平行边、简单图、完全图、

连通图、回路、子图、树以及生成树。

28.3 具有 5 个顶点的完全图中有几条边？具有 5 个结点的树中有几条边？

28.4 具有 n 个顶点的完全图中有几条边？具有 n 个结点的树中有几条边？

28.3 表示图

要点提示：表示图是在程序中存储它的顶点和边。存储图的数据结构是数组或者线性表。为了编写处理和操作图的程序，必须在计算机中存储和表示图。

28.3.1 表示顶点

顶点可以存储在数组或线性表中。例如，图 28-1 中的所有城市名可以用下面的数组来存储：

```
String[] vertices = {"Seattle", "San Francisco", "Los Angeles",  
    "Denver", "Kansas City", "Chicago", "Boston", "New York",  
    "Atlanta", "Miami", "Dallas", "Houston"};
```

注意：顶点可以是任意类型的对象。例如，可以将城市考虑为包含名字、人口和市长等信息的实体。于是，可以将顶点定义为：

```
City city0 = new City("Seattle", 608660, "Mike McGinn");  
...  
City city11 = new City("Houston", 2099451, "Annise Parker");  
City[] vertices = {city0, city1, ..., city11};
```

```
public class City {  
    private String cityName;  
    private int population;  
    private String mayor;  
  
    public City(String cityName, int population, String mayor) {  
        this.cityName = cityName;  
        this.population = population;  
        this.mayor = mayor;  
    }  
  
    public String getCityName() {  
        return cityName;  
    }  
  
    public int getPopulation() {  
        return population;  
    }  
  
    public String getMayor() {  
        return mayor;  
    }  
  
    public void setMayor(String mayor) {  
        this.mayor = mayor;  
    }  
  
    public void setPopulation(int population) {  
        this.population = population;  
    }  
}
```

对于一个拥有 n 个顶点的图，这 n 个顶点可以使用自然数 $0, 1, 2, \dots, n-1$ 来标注。于是，`vertices[0]` 表示 "Seattle"，`vertices[1]` 表示 "San Francisco"，等等，如图 28-5 所示。

vertices[0]	Seattle
vertices[1]	San Francisco
vertices[2]	Los Angeles
vertices[3]	Denver
vertices[4]	Kansas City
vertices[5]	Chicago
vertices[6]	Boston
vertices[7]	New York
vertices[8]	Atlanta
vertices[9]	Miami
vertices[10]	Dallas
vertices[11]	Houston

图 28-5 存储顶点名字的数组

注意：可以通过顶点的名字或者索引来引用顶点，就看哪一种方式使用起来更方便。很显然，在程序中通过索引访问顶点是很容易的。

28.3.2 表示边：边数组

边可以使用二维数组来表示。例如，可以使用下面的数组来存储图 28-1 中图的所有边：

```
int[][] edges = {
    {0, 1}, {0, 3}, {0, 5},
    {1, 0}, {1, 2}, {1, 3},
    {2, 1}, {2, 3}, {2, 4}, {2, 10},
    {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
    {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
    {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
    {6, 5}, {6, 7},
    {7, 4}, {7, 5}, {7, 6}, {7, 8},
    {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
    {9, 8}, {9, 11},
    {10, 2}, {10, 4}, {10, 8}, {10, 11},
    {11, 8}, {11, 9}, {11, 10}
};
```

这就是所谓的边数组（edge array）。图 28-3 中的顶点和边可以如下表示：

```
String[] names = {"Peter", "Jane", "Mark", "Cindy", "Wendy"};

int[][] edges = {{0, 2}, {1, 2}, {2, 4}, {3, 4}};
```

28.3.3 表示边：Edge 对象

另外一种表示边的方法就是将边定义为对象，并存储在 `java.util.ArrayList` 中。Edge 类可以如下定义：

```
public class Edge {
    int u;
    int v;

    public Edge(int u, int v) {
```

```

    this.u = u;
    this.v = v;
}

public boolean equals(Object o) {
    return u == ((Edge)o).u && v == ((Edge)o).v;
}
}

```

例如，可以使用下面的线性表来存储图 28-1 中图的所有边：

```

java.util.ArrayList<Edge> list = new java.util.ArrayList<>();
list.add(new Edge(0, 1));
list.add(new Edge(0, 3));
list.add(new Edge(0, 5));
...

```

如果事先不知道所有的边，那么将 Edge 对象存储在一个 ArrayList 中是很有用的。

使用 28.3.2 节中的边数组和本节前面的 Edge 对象来表示边对输入来说是很直观的，但是内部处理的效率不高。接下来的两节将介绍使用邻接矩阵（adjacency matrice）和邻接线性表（adjacency list）来表示图，使用这两种数据结构处理图很高效。

28.3.4 表示边：邻接矩阵

假设图有 n 个顶点，那么可以使用名为 adjacencyMatrix 的二维 $n \times n$ 矩阵来表示边。矩阵中的每一个元素或者为 0 或者为 1。如果从顶点 i 到顶点 j 存在一条边，那么 adjacencyMatrix[i][j] 为 1；否则，adjacencyMatrix[i][j] 为 0。如果图是无向的，由于 adjacencyMatrix[i][j] 与 adjacencyMatrix[j][i] 是相同的，所以矩阵是对称的。例如，图 28-1 中图的边可以使用邻接矩阵表示为：

```

int[][] adjacencyMatrix = {
    {0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0}, // Seattle
    {1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0}, // San Francisco
    {0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0}, // Los Angeles
    {1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0}, // Denver
    {0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0}, // Kansas City
    {1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0}, // Chicago
    {0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0}, // Boston
    {0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0}, // New York
    {0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1}, // Atlanta
    {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1}, // Miami
    {0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1}, // Dallas
    {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0} // Houston
};

```

注意：由于对于无向图来说，矩阵是对称的，因此可以用锯齿矩阵来存储它。

图 28-3a 中的有向图的邻接矩阵可以如下表示：

```

int[][] a = {{0, 0, 1, 0, 0}, // Peter
             {0, 0, 1, 0, 0}, // Jane
             {0, 0, 0, 0, 1}, // Mark
             {0, 0, 0, 0, 1}, // Cindy
             {0, 0, 0, 0, 0} // Wendy
};

```

28.3.5 表示边：邻接线性表

可以使用邻接顶点线性表（adjacency vertex list）和邻接边线性表（adjacency edge list）来表示边。顶点 i 的邻接顶点线性表包含了所有与 i 有边相连的顶点。顶点 i 的邻接边线性

表包含了所有与 i 相连的边。可以定义一个线性表数组。数组具有 n 个条目，每个条目是一个线性表。顶点 i 的邻接顶点线性表包含了所有的顶点 j ，其中顶点 i 和 j 之间有一条边。例如，为了表示图 28-1 中的图，可以如下创建一个线性表数组：

```
java.util.List<Integer>[] neighbors = new java.util.List[12];
```

`neighbors[0]` 包含顶点 0（即 Seattle）的所有邻接顶点，`neighbors [1]` 包含顶点 1（即 San Francisco）的所有邻接顶点，以此类推，如图 28-6 所示。

Seattle	neighbors[0]	1	3	5				
San Francisco	neighbors[1]	0	2	3				
Los Angeles	neighbors[2]	1	3	4	10			
Denver	neighbors[3]	0	1	2	4	5		
Kansas City	neighbors[4]	2	3	5	7	8	10	
Chicago	neighbors[5]	0	3	4	6	7		
Boston	neighbors[6]	5	7					
New York	neighbors[7]	4	5	6	8			
Atlanta	neighbors[8]	4	7	9	10	11		
Miami	neighbors[9]	8	11					
Dallas	neighbors[10]	2	4	8	11			
Houston	neighbors[11]	8	9	10				

图 28-6 图 28-1 中图的边使用邻接顶点线性表表示

为了表示图 28-1 中图的邻接边线性表，可以如下创建一个线性表数组：

```
java.util.List<Edge>[] neighbors = new java.util.List[12];
```

`neighbors [0]` 包含顶点 0（即 Seattle）的所有邻接边，`neighbors [1]` 包含顶点 1（即 San Francisco）的所有邻接边，以此类推，如图 28-7 所示。

Seattle	neighbors[0]	Edge(0, 1)	Edge(0, 3)	Edge(0, 5)			
San Francisco	neighbors[1]	Edge(1, 0)	Edge(1, 2)	Edge(1, 3)			
Los Angeles	neighbors[2]	Edge(2, 1)	Edge(2, 3)	Edge(2, 4)	Edge(2, 10)		
Denver	neighbors[3]	Edge(3, 0)	Edge(3, 1)	Edge(3, 2)	Edge(3, 4)	Edge(3, 5)	
Kansas City	neighbors[4]	Edge(4, 2)	Edge(4, 3)	Edge(4, 5)	Edge(4, 7)	Edge(4, 8)	Edge(4, 10)
Chicago	neighbors[5]	Edge(5, 0)	Edge(5, 3)	Edge(5, 4)	Edge(5, 6)	Edge(5, 7)	
Boston	neighbors[6]	Edge(6, 5)	Edge(6, 7)				
New York	neighbors[7]	Edge(7, 4)	Edge(7, 5)	Edge(7, 6)	Edge(7, 8)		
Atlanta	neighbors[8]	Edge(8, 4)	Edge(8, 7)	Edge(8, 9)	Edge(8, 10)	Edge(8, 11)	
Miami	neighbors[9]	Edge(9, 8)	Edge(9, 11)				
Dallas	neighbors[10]	Edge(10, 2)	Edge(10, 4)	Edge(10, 8)	Edge(10, 11)		
Houston	neighbors[11]	Edge(11, 8)	Edge(11, 9)	Edge(11, 10)			

图 28-7 图 28-1 中图的边使用邻接边线性表表示

注意：可以使用邻接矩阵或者邻接线性表来表示一个图。哪种方法更好呢？如果图很密

(也就是说, 存在大量的边), 那么建议使用邻接矩阵。如果图很稀疏(也就是说, 存在很少的边), 由于使用邻接矩阵会浪费大量的存储空间, 因此最好使用邻接线性表。

邻接矩阵和邻接线性表都可以用在程序中, 以使算法的效率更高。例如, 使用邻接矩阵来检查两个顶点是否相连只需要 $O(1)$ 常量时间, 而使用邻接线性表来打印图中所有的边需要线性时间 $O(m)$, 这里的 m 表示边的条数。

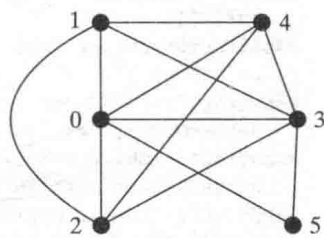
注意: 用邻接顶点线性表表示无权图更加简单。然而, 对于许多应用来说, 邻接边线性表更加灵活。使用邻接边线性表更易于在边上添加额外的约束。出于这个原因, 本书将用邻接边线性表来表示图。

可以使用数组、数组线性表或者链表来存储邻接线性表。我们使用线性表而不使用数组, 因为线性表更易于扩充来添加新的顶点。而且我们使用数组线性表而不是链表, 因为我们的算法仅要求搜索线性表中的邻接顶点。对于我们的算法而言, 使用数组线性表更加高效。使用数组线性表, 图 28-6 中的邻接边线性表可以如下构建:

```
List<ArrayList<Edge>> neighbors = new ArrayList<>();
neighbors.add(new ArrayList<Edge>());
neighbors.get(0).add(new Edge(0, 1));
neighbors.get(0).add(new Edge(0, 3));
neighbors.get(0).add(new Edge(0, 5));
neighbors.add(new ArrayList<Edge>());
neighbors.get(1).add(new Edge(1, 0));
neighbors.get(1).add(new Edge(1, 2));
neighbors.get(1).add(new Edge(1, 3));
...
neighbors.get(11).add(new Edge(11, 8));
neighbors.get(11).add(new Edge(11, 9));
neighbors.get(11).add(new Edge(11, 10));
```

复习题

- 28.5 如何表示图中的顶点? 如何使用边数组来表示边? 如何使用边对象来表示边? 如何使用邻接矩阵来表示边? 如何使用邻接线性表来表示边?
- 28.6 分别使用边数组、边对象线性表、邻接矩阵、邻接顶点线性表、邻接边线性表表示下面的图。



28.4 图建模

要点提示: Graph 接口定义了图的通用操作。

Java 合集框架是设计复杂的数据结构的好示例。数据结构的常用特征在接口中定义(例如, Collection、Set、List、Queue), 如图 20-1 所示。抽象类(例如, AbstractCollection、AbstractSet、AbstractList)部分地实现了这个接口。具体类(例如, HashSet、LinkedHashSet、TreeSet、ArrayList、LinkedList、PriorityQueue)提供了具体的实现。这种设计模式对建模图非常有用。我们将定义一个名为 Graph 的接口来包含图的所有常用操作, 以及一个名为 AbstractGraph 的抽象类来部分地实现 Graph 接口。许多具体的图被添加到这个设计中。例如, 我们将定义这样的名为 UnweightedGraph 和 WeightedGraph 的图。这些接口和类的关系如图 28-8 所示。

什么是图的常用操作? 一般来说, 需要得到图中顶点的个数, 得到图中所有的顶点, 得到指定下标的顶点对象, 得到指定名字的顶点的下标, 得到顶点的邻居, 得到顶点的度, 清除图, 添加新的顶点, 添加新的边, 执行深度优先搜索及广度优先搜索。深度优先搜索及广

度优先搜索将在下一节中介绍。图 28-9 在 UML 图中列举出这些方法。

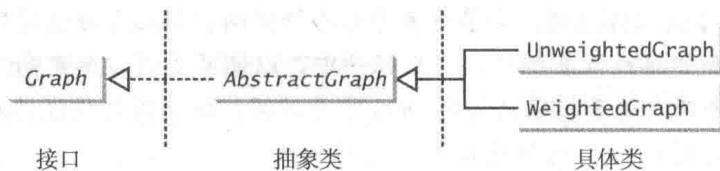


图 28-8 使用接口、抽象类和具体类建模图

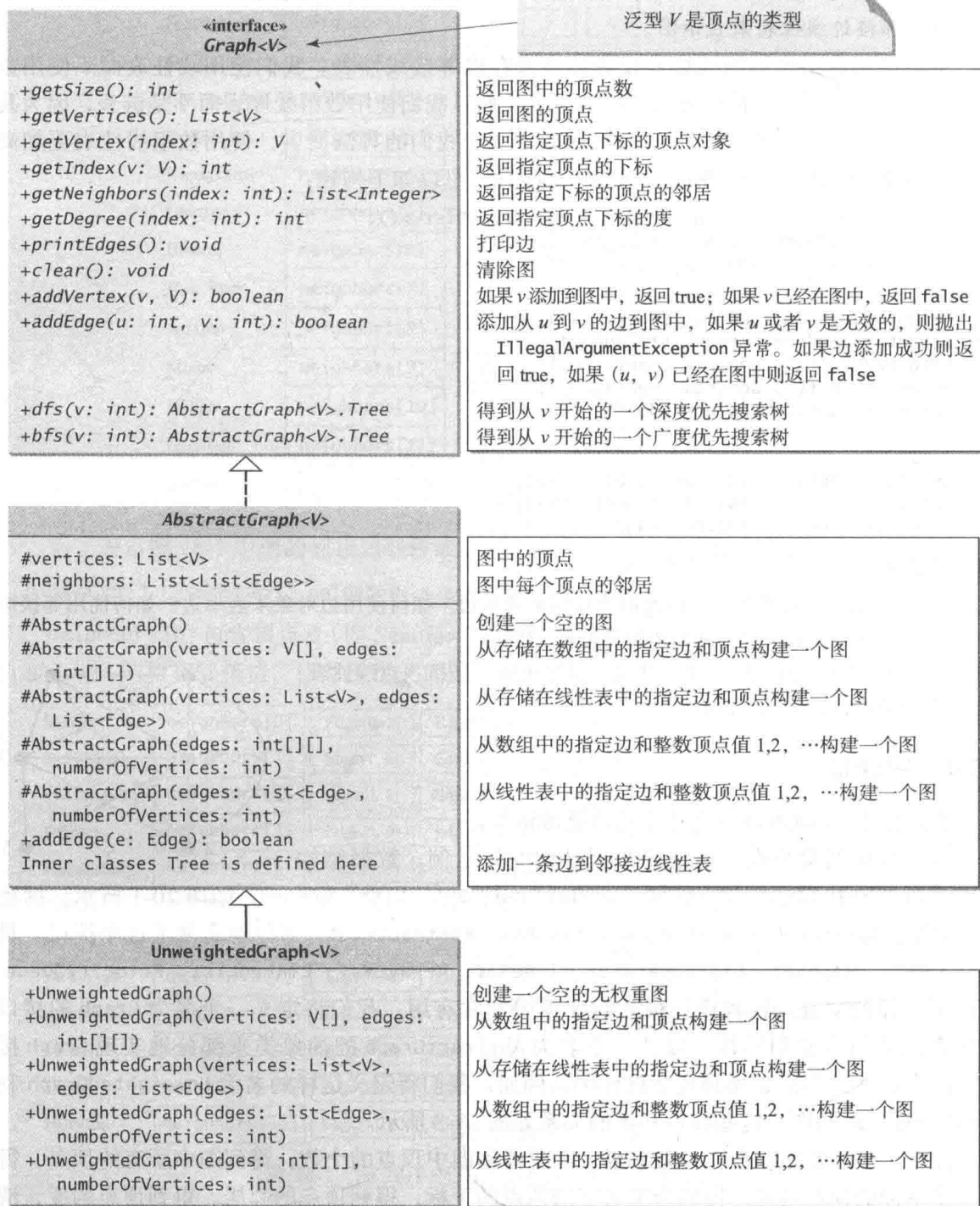


图 28-9 Graph 接口定义所有类型的图的常用操作

AbstractGraph 没有引入任何新方法。在 AbstractGraph 类中定义了一个顶点的线性表和一个边的邻接线性表。有了这些数据域，就足够实现所有定义在 Graph 接口中的方法。方便起见，假设图是简单图，即顶点没有到自身的边，没有从顶点 u 到 v 的平行边。

AbstractGraph 类实现了 Graph 接口的所有方法，除了一个便于添加一个 Edge 对象到邻接边线性表的 addEdge(edge) 方法外，它没有引入任何新的方法。UnweightedGraph 简单地继承了 AbstractGraph，它用 5 个构造方法创建具体的 Graph 实例。

{ } 注意：可以使用任意类型的顶点来创建图。每个顶点与一个下标相关联，该下标同顶点线性表中的顶点下标是一样的。如果创建图时没有指定顶点，顶点和它们的索引一样。

{ } 注意：AbstractGraph 类实现了 Graph 接口的所有方法，那么为什么将它定义为抽象的？今后，我们可能需要给 Graph 接口添加 AbstractGraph 不能实现的新方法。为了使类易于维护，将 AbstractGraph 定义为抽象类会比较合适。

假设所有这些接口和类都是可用的。程序清单 28-1 给出了一个测试程序，它为图 28-1 创建一个图，并且为图 28-3a 创建一个图。

程序清单 28-1 TestGraph.java

```

1 public class TestGraph {
2     public static void main(String[] args) {
3         String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
4                               "Denver", "Kansas City", "Chicago", "Boston", "New York",
5                               "Atlanta", "Miami", "Dallas", "Houston"};
6
7         // Edge array for graph in Figure 28.1
8         int[][] edges = {
9             {0, 1}, {0, 3}, {0, 5},
10            {1, 0}, {1, 2}, {1, 3},
11            {2, 1}, {2, 3}, {2, 4}, {2, 10},
12            {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
13            {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
14            {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
15            {6, 5}, {6, 7},
16            {7, 4}, {7, 5}, {7, 6}, {7, 8},
17            {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
18            {9, 8}, {9, 11},
19            {10, 2}, {10, 4}, {10, 8}, {10, 11},
20            {11, 8}, {11, 9}, {11, 10}
21        };
22
23        Graph<String> graph1 = new UnweightedGraph<>(vertices, edges);
24        System.out.println("The number of vertices in graph1: "
25            + graph1.getSize());
26        System.out.println("The vertex with index 1 is "
27            + graph1.getVertex(1));
28        System.out.println("The index for Miami is " +
29            graph1.getIndex("Miami"));
30        System.out.println("The edges for graph1:");
31        graph1.printEdges();
32
33        // List of Edge objects for graph in Figure 28.3a
34        String[] names = {"Peter", "Jane", "Mark", "Cindy", "Wendy"};
35        java.util.ArrayList<AbstractGraph.Edge> edgeList
36            = new java.util.ArrayList<>();
37        edgeList.add(new AbstractGraph.Edge(0, 2));
38        edgeList.add(new AbstractGraph.Edge(1, 2));
39        edgeList.add(new AbstractGraph.Edge(2, 4));
40        edgeList.add(new AbstractGraph.Edge(3, 4));
41        // Create a graph with 5 vertices

```

```

42     Graph<String> graph2 = new UnweightedGraph<>
43         (java.util.Arrays.asList(names), edgeList);
44     System.out.println("\nThe number of vertices in graph2: "
45         + graph2.getSize());
46     System.out.println("The edges for graph2:");
47     graph2.printEdges();
48 }
49 }

```

```

The number of vertices in graph1: 12
The vertex with index 1 is San Francisco
The index for Miami is 9
The edges for graph1:
Seattle (0): (0, 1) (0, 3) (0, 5)
San Francisco (1): (1, 0) (1, 2) (1, 3)
Los Angeles (2): (2, 1) (2, 3) (2, 4) (2, 10)
Denver (3): (3, 0) (3, 1) (3, 2) (3, 4) (3, 5)
Kansas City (4): (4, 2) (4, 3) (4, 5) (4, 7) (4, 8) (4, 10)
Chicago (5): (5, 0) (5, 3) (5, 4) (5, 6) (5, 7)
Boston (6): (6, 5) (6, 7)
New York (7): (7, 4) (7, 5) (7, 6) (7, 8)
Atlanta (8): (8, 4) (8, 7) (8, 9) (8, 10) (8, 11)
Miami (9): (9, 8) (9, 11)
Dallas (10): (10, 2) (10, 4) (10, 8) (10, 11)
Houston (11): (11, 8) (11, 9) (11, 10)

The number of vertices in graph2: 5
The edges for graph2:
Peter (0): (0, 2)
Jane (1): (1, 2)
Mark (2): (2, 4)
Cindy (3): (3, 4)
Wendy (4):

```

程序在第 3 ~ 23 行为图 28-1 中的图创建 graph1。graph1 中的顶点在第 3 ~ 5 行定义。graph1 的边在第 8 ~ 21 行定义。这里使用二维数组来表示边。对于数组中的每一行 i，edges[i][0] 和 edges[i][1] 表示存在从顶点 edges[i][0] 到顶点 edges[i][1] 的一条边。例如，第一行 {0,1} 表示从顶点 0(edges[0][0]) 到顶点 1(edges[0][1]) 的边，行 {0,5} 表示从顶点 0(edges[2][0]) 到顶点 5(edges[2][1]) 的边。第 23 行创建图。第 31 行调用 graph1 上的方法 printEdges() 来显示 graph1 中的所有边。

程序在第 34 ~ 43 行为图 28-3a 中的图创建 graph2。第 37 ~ 40 行定义 graph2 中的边。第 43 行使用 Edge 对象的线性表创建 graph2。第 47 行调用 graph2 上的方法 printEdges() 来显示 graph2 中的所有边。

注意，graph1 和 graph2 都包含字符串顶点。这些顶点与下标 0,1,...,n-1 相关联。下标是顶点在 vertices 中的位置。例如，顶点 Miami 的下标是 9。

现在将注意力放在实现接口和类上。程序清单 28-2、程序清单 28-3 和程序清单 28-4 分别给出 Graph 接口、AbstractGraph 类以及 UnweightedGraph 类的具体实现。

程序清单 28-2 Graph.java

```

1 public interface Graph<V> {
2     /** Return the number of vertices in the graph */
3     public int getSize();
4
5     /** Return the vertices in the graph */
6     public java.util.List<V> getVertices();
7

```

```

8  /** Return the object for the specified vertex index */
9  public V getVertex(int index);
10
11 /** Return the index for the specified vertex object */
12 public int getIndex(V v);
13
14 /** Return the neighbors of vertex with the specified index */
15 public java.util.List<Integer> getNeighbors(int index);
16
17 /** Return the degree for a specified vertex */
18 public int getDegree(int v);
19
20 /** Print the edges */
21 public void printEdges();
22
23 /** Clear the graph */
24 public void clear();
25
26 /** Add a vertex to the graph */
27 public void addVertex(V vertex);
28
29 /** Add an edge to the graph */
30 public void addEdge(int u, int v);
31
32 /** Obtain a depth-first search tree starting from v */
33 public AbstractGraph<V>.Tree dfs(int v);
34
35 /** Obtain a breadth-first search tree starting from v */
36 public AbstractGraph<V>.Tree bfs(int v);
37 }

```

程序清单 28-3 AbstractGraph.java

```

1  import java.util.*;
2
3  public abstract class AbstractGraph<V> implements Graph<V> {
4      protected List<V> vertices = new ArrayList<>(); // Store vertices
5      protected List<List<Edge>> neighbors
6          = new ArrayList<>(); // Adjacency lists
7
8      /** Construct an empty graph */
9      protected AbstractGraph() {
10     }
11
12     /** Construct a graph from vertices and edges stored in arrays */
13     protected AbstractGraph(V[] vertices, int[][] edges) {
14         for (int i = 0; i < vertices.length; i++)
15             addVertex(vertices[i]);
16
17         createAdjacencyLists(edges, vertices.length);
18     }
19
20     /** Construct a graph from vertices and edges stored in List */
21     protected AbstractGraph(List<V> vertices, List<Edge> edges) {
22         for (int i = 0; i < vertices.size(); i++)
23             addVertex(vertices.get(i));
24
25         createAdjacencyLists(edges, vertices.size());
26     }
27
28     /** Construct a graph for integer vertices 0, 1, 2 and edge list */
29     protected AbstractGraph(List<Edge> edges, int numberOfVertices) {
30         for (int i = 0; i < numberOfVertices; i++)
31             addVertex((V)(new Integer(i))); // vertices is {0, 1, ...}

```

```

32     createAdjacencyLists(edges, numberOfVertices);
33 }
34
35
36 /** Construct a graph from integer vertices 0, 1, and edge array */
37 protected AbstractGraph(int[][] edges, int numberOfVertices) {
38     for (int i = 0; i < numberOfVertices; i++)
39         addVertex((V)(new Integer(i))); // vertices is {0, 1, ...}
40
41     createAdjacencyLists(edges, numberOfVertices);
42 }
43
44 /** Create adjacency lists for each vertex */
45 private void createAdjacencyLists(
46     int[][] edges, int numberOfVertices) {
47     for (int i = 0; i < edges.length; i++) {
48         addEdge(edges[i][0], edges[i][1]);
49     }
50 }
51
52 /** Create adjacency lists for each vertex */
53 private void createAdjacencyLists(
54     List<Edge> edges, int numberOfVertices) {
55     for (Edge edge: edges) {
56         addEdge(edge.u, edge.v);
57     }
58 }
59
60 @Override /** Return the number of vertices in the graph */
61 public int getSize() {
62     return vertices.size();
63 }
64
65 @Override /** Return the vertices in the graph */
66 public List<V> getVertices() {
67     return vertices;
68 }
69
70 @Override /** Return the object for the specified vertex */
71 public V getVertex(int index) {
72     return vertices.get(index);
73 }
74
75 @Override /** Return the index for the specified vertex object */
76 public int getIndex(V v) {
77     return vertices.indexOf(v);
78 }
79
80 @Override /** Return the neighbors of the specified vertex */
81 public List<Integer> getNeighbors(int index) {
82     List<Integer> result = new ArrayList<>();
83     for (Edge e: neighbors.get(index))
84         result.add(e.v);
85
86     return result;
87 }
88
89 @Override /** Return the degree for a specified vertex */
90 public int getDegree(int v) {
91     return neighbors.get(v).size();
92 }
93
94 @Override /** Print the edges */
95 public void printEdges() {

```

```
96     for (int u = 0; u < neighbors.size(); u++) {
97         System.out.print(getVertex(u) + " (" + u + "): ");
98         for (Edge e: neighbors.get(u)) {
99             System.out.print("(" + getVertex(e.u) + ", " +
100                getVertex(e.v) + ") ");
101         }
102         System.out.println();
103     }
104 }
105
106 @Override /** Clear the graph */
107 public void clear() {
108     vertices.clear();
109     neighbors.clear();
110 }
111
112 @Override /** Add a vertex to the graph */
113 public boolean addVertex(V vertex) {
114     if (!vertices.contains(vertex)) {
115         vertices.add(vertex);
116         neighbors.add(new ArrayList<Edge>());
117         return true;
118     }
119     else {
120         return false;
121     }
122 }
123
124 /** Add an edge to the graph */
125 protected boolean addEdge(Edge e) {
126     if (e.u < 0 || e.u > getSize() - 1)
127         throw new IllegalArgumentException("No such index: " + e.u);
128
129     if (e.v < 0 || e.v > getSize() - 1)
130         throw new IllegalArgumentException("No such index: " + e.v);
131
132     if (!neighbors.get(e.u).contains(e)) {
133         neighbors.get(e.u).add(e);
134         return true;
135     }
136     else {
137         return false;
138     }
139 }
140
141 @Override /** Add an edge to the graph */
142 public boolean addEdge(int u, int v) {
143     return addEdge(new Edge(u, v));
144 }
145
146 /** Edge inner class inside the AbstractGraph class */
147 public static class Edge {
148     public int u; // Starting vertex of the edge
149     public int v; // Ending vertex of the edge
150
151     /** Construct an edge for (u, v) */
152     public Edge(int u, int v) {
153         this.u = u;
154         this.v = v;
155     }
156
157     public boolean equals(Object o) {
158         return u == ((Edge)o).u && v == ((Edge)o).v;
159     }
160 }
```



```

160     }
161
162     @Override /** Obtain a DFS tree starting from vertex v */
163     /** To be discussed in Section 28.7 */
164     public Tree dfs(int v) {
165         List<Integer> searchOrder = new ArrayList<>();
166         int[] parent = new int[vertices.size()];
167         for (int i = 0; i < parent.length; i++)
168             parent[i] = -1; // Initialize parent[i] to -1
169
170         // Mark visited vertices
171         boolean[] isVisited = new boolean[vertices.size()];
172
173         // Recursively search
174         dfs(v, parent, searchOrder, isVisited);
175
176         // Return a search tree
177         return new Tree(v, parent, searchOrder);
178     }
179
180     /** Recursive method for DFS search */
181     private void dfs(int u, int[] parent, List<Integer> searchOrder,
182                     boolean[] isVisited) {
183         // Store the visited vertex
184         searchOrder.add(u);
185         isVisited[u] = true; // Vertex v visited
186
187         for (Edge e : neighbors.get(u)) {
188             if (!isVisited[e.v]) {
189                 parent[e.v] = u; // The parent of vertex e.v is u
190                 dfs(e.v, parent, searchOrder, isVisited); // Recursive search
191             }
192         }
193     }
194
195     @Override /** Starting bfs search from vertex v */
196     /** To be discussed in Section 28.9 */
197     public Tree bfs(int v) {
198         List<Integer> searchOrder = new ArrayList<>();
199         int[] parent = new int[vertices.size()];
200         for (int i = 0; i < parent.length; i++)
201             parent[i] = -1; // Initialize parent[i] to -1
202
203         java.util.LinkedList<Integer> queue =
204             new java.util.LinkedList<>(); // list used as a queue
205         boolean[] isVisited = new boolean[vertices.size()];
206         queue.offer(v); // Enqueue v
207         isVisited[v] = true; // Mark it visited
208
209         while (!queue.isEmpty()) {
210             int u = queue.poll(); // Dequeue to u
211             searchOrder.add(u); // u searched
212             for (Edge e : neighbors.get(u)) {
213                 if (!isVisited[e.v]) {
214                     queue.offer(e.v); // Enqueue w
215                     parent[e.v] = u; // The parent of w is u
216                     isVisited[e.v] = true; // Mark it visited
217                 }
218             }
219         }
220
221         return new Tree(v, parent, searchOrder);
222     }
223
224     /** Tree inner class inside the AbstractGraph class */

```

```

225  /** To be discussed in Section 28.6 */
226  public class Tree {
227      private int root; // The root of the tree
228      private int[] parent; // Store the parent of each vertex
229      private List<Integer> searchOrder; // Store the search order
230
231      /** Construct a tree with root, parent, and searchOrder */
232      public Tree(int root, int[] parent, List<Integer> searchOrder) {
233          this.root = root;
234          this.parent = parent;
235          this.searchOrder = searchOrder;
236      }
237
238      /** Return the root of the tree */
239      public int getRoot() {
240          return root;
241      }
242
243      /** Return the parent of vertex v */
244      public int getParent(int v) {
245          return parent[v];
246      }
247
248      /** Return an array representing search order */
249      public List<Integer> getSearchOrder() {
250          return searchOrder;
251      }
252
253      /** Return number of vertices found */
254      public int getNumberOfVerticesFound() {
255          return searchOrder.size();
256      }
257
258      /** Return the path of vertices from a vertex to the root */
259      public List<V> getPath(int index) {
260          ArrayList<V> path = new ArrayList<>();
261
262          do {
263              path.add(vertices.get(index));
264              index = parent[index];
265          }
266          while (index != -1);
267
268          return path;
269      }
270
271      /** Print a path from the root to vertex v */
272      public void printPath(int index) {
273          List<V> path = getPath(index);
274          System.out.print("A path from " + vertices.get(root) + " to " +
275              vertices.get(index) + ": ");
276          for (int i = path.size() - 1; i >= 0; i--)
277              System.out.print(path.get(i) + " ");
278      }
279
280      /** Print the whole tree */
281      public void printTree() {
282          System.out.println("Root is: " + vertices.get(root));
283          System.out.print("Edges: ");
284          for (int i = 0; i < parent.length; i++) {
285              if (parent[i] != -1) {
286                  // Display an edge
287                  System.out.print("(" + vertices.get(parent[i]) + ", " +
288                      vertices.get(i) + ") ");
289              }

```

```

290     }
291     System.out.println();
292 }
293 }
294 }

```

程序清单 28-4 UnweightedGraph.java

```

1  import java.util.*;
2
3  public class UnweightedGraph<V> extends AbstractGraph<V> {
4      /** Construct an empty graph */
5      public UnweightedGraph() {
6      }
7
8      /** Construct a graph from vertices and edges stored in arrays */
9      public UnweightedGraph(V[] vertices, int[][] edges) {
10         super(vertices, edges);
11     }
12
13     /** Construct a graph from vertices and edges stored in List */
14     public UnweightedGraph(List<V> vertices, List<Edge> edges) {
15         super(vertices, edges);
16     }
17
18     /** Construct a graph for integer vertices 0, 1, 2 and edge list */
19     public UnweightedGraph(List<Edge> edges, int numberOfVertices) {
20         super(edges, numberOfVertices);
21     }
22
23     /** Construct a graph from integer vertices 0, 1, and edge array */
24     public UnweightedGraph(int[][] edges, int numberOfVertices) {
25         super(edges, numberOfVertices);
26     }
27 }

```

程序清单 28-2 中的 Graph 接口的代码和程序清单 28-4 中 UnweightedGraph 类的代码都很简单易懂。下面消化一下程序清单 28-3 中的 AbstractGraph 类的代码。

AbstractGraph 类定义了数据域 vertices (第 4 行) 来存储顶点, 定义 neighbors (第 5 行) 来在邻接线性表中存储边。neighbors.get(i) 存储所有与顶点 i 相连的顶点。在第 9 ~ 42 行定义 4 个重载的构造方法, 可以创建默认图, 或者从边和顶点的数组或者线性表来创建图。方法 createAdjacencyLists(int[][] edges, int numberOfVertices) 从一个数组中的边创建邻接线性表 (第 45 ~ 50 行)。方法 createAdjacencyLists(List<Edge> edges, int numberOfVertices) 从一个线性表中的边创建邻接线性表 (第 53 ~ 58 行)。

getNeighbors(u) 方法 (第 81 ~ 87 行) 返回顶点 u 的邻接顶点线性表。clear() 方法 (第 106 ~ 110 行) 从图中删除所有顶点和边。addVertex(u) 方法 (第 112 ~ 122 行) 添加一个新的顶点到 vertices 并返回 true。如果顶点已经在图中了则返回 false (第 120 行)。

addEdge(e) 方法 (第 124 ~ 139 行) 添加一个新的边到邻接边线性表中并返回 true。如果边已经在图中了则返回 false。如果边是无效的, 则该方法可能抛出 IllegalArgumentException (第 126 ~ 130 行)。

方法 printEdges() (第 95 ~ 104 行) 显示了所有的顶点以及与每一个顶点相连的边。

第 164 ~ 293 行的代码给出查找深度优先搜索树和广度优先搜索树的方法, 这将在 28.7 节和 28.9 节中介绍。

✓ 复习题

28.7 描述 Graph、AbstractGraph 和 UnweightedGraph 之间的关系。

28.8 对于程序清单 28-1 中的代码来说，什么是 `graph1.getIndex("Seattle")`？什么是 `graph1.getDegree(5)`？什么是 `graph1.getVertex(4)`？

28.5 图的可视化

🔑 要点提示：为了可视化地显示图，每个顶点必须赋予一个位置。

前一节介绍了如何使用 Graph 接口、AbstractGraph 类和 UnweightedGraph 类来对图建模。本节介绍如何用图形显示图。为了显示一个图，需要知道每个顶点的位置以及名字。为了确保图可以显示，我们在程序清单 28-5 中定义一个名为 Displayable 的接口，该接口具有获取 x 和 y 坐标以及顶点的名字，并且让顶点作为 Displayable 的实例。

程序清单 28-5 Displayable.java

```
1 public interface Displayable {
2     public int getX(); // Get x-coordinate of the vertex
3     public int getY(); // Get y-coordinate of the vertex
4     public String getName(); // Get display name of the vertex
5 }
```

现在，一个带 Displayable 顶点的图可以显示在一个名为 GraphView 的面板上，如程序清单 28-6 所示。

程序清单 28-6 GraphView.java

```
1 import javafx.scene.layout.Pane;
2 import javafx.scene.shape.Circle;
3 import javafx.scene.shape.Line;
4 import javafx.scene.text.Text;
5
6 public class GraphView extends Pane {
7     private Graph<? extends Displayable> graph;
8
9     public GraphView(Graph<? extends Displayable> graph) {
10         this.graph = graph;
11
12         // Draw vertices
13         java.util.List<? extends Displayable> vertices
14             = graph.getVertices();
15         for (int i = 0; i < graph.getSize(); i++) {
16             int x = vertices.get(i).getX();
17             int y = vertices.get(i).getY();
18             String name = vertices.get(i).getName();
19
20             getChildren().add(new Circle(x, y, 16)); // Display a vertex
21             getChildren().add(new Text(x - 8, y - 18, name));
22         }
23
24         // Draw edges for pairs of vertices
25         for (int i = 0; i < graph.getSize(); i++) {
26             java.util.List<Integer> neighbors = graph.getNeighbors(i);
27             int x1 = graph.getVertex(i).getX();
28             int y1 = graph.getVertex(i).getY();
29             for (int v: neighbors) {
30                 int x2 = graph.getVertex(v).getX();
31                 int y2 = graph.getVertex(v).getY();
32
33                 // Draw an edge for (i, v)
34                 getChildren().add(new Line(x1, y1, x2, y2));
```

```

35     }
36 }
37 }
38 }

```

要在面板上显示图，只需通过将图作为参数传入构造方法来创建一个 `GraphView` 的实例（第 9 行）。要显示顶点，图顶点的类必须实现 `Displayable` 接口（第 13 ~ 22 行）。对于每个顶点的下标 *i*，调用方法 `graph.getNeighbors(i)` 返回它的邻接线性表（第 26 行）。通过这个线性表，可以发现所有与顶点 *i* 相邻的顶点，并且绘出一条将顶点 *i* 与其相邻顶点相连的线（第 27 ~ 34 行）。

程序清单 27-7 给出一个显示图 28-1 中图的例子，如图 28-10 所示。

程序清单 28-7 DisplayUSMap.java

```

1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.stage.Stage;
4
5  public class DisplayUSMap extends Application {
6      @Override // Override the start method in the Application class
7      public void start(Stage primaryStage) {
8          City[] vertices = {new City("Seattle", 75, 50),
9                          new City("San Francisco", 50, 210),
10                         new City("Los Angeles", 75, 275), new City("Denver", 275, 175),
11                         new City("Kansas City", 400, 245),
12                         new City("Chicago", 450, 100), new City("Boston", 700, 80),
13                         new City("New York", 675, 120), new City("Atlanta", 575, 295),
14                         new City("Miami", 600, 400), new City("Dallas", 408, 325),
15                         new City("Houston", 450, 360) };
16
17          // Edge array for graph in Figure 28.1
18          int[][] edges = {
19              {0, 1}, {0, 3}, {0, 5}, {1, 0}, {1, 2}, {1, 3},
20              {2, 1}, {2, 3}, {2, 4}, {2, 10},
21              {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
22              {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
23              {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
24              {6, 5}, {6, 7}, {7, 4}, {7, 5}, {7, 6}, {7, 8},
25              {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
26              {9, 8}, {9, 11}, {10, 2}, {10, 4}, {10, 8}, {10, 11},
27              {11, 8}, {11, 9}, {11, 10}
28          };
29
30          Graph<City> graph = new UnweightedGraph<>(vertices, edges);
31
32          // Create a scene and place it in the stage
33          Scene scene = new Scene(new GraphView(graph), 750, 450);
34          primaryStage.setTitle("DisplayUSMap"); // Set the stage title
35          primaryStage.setScene(scene); // Place the scene in the stage
36          primaryStage.show(); // Display the stage
37      }
38
39      static class City implements Displayable {
40          private int x, y;
41          private String name;
42
43          City(String name, int x, int y) {
44              this.name = name;
45              this.x = x;
46              this.y = y;
47          }
48      }

```

```

49     @Override
50     public int getX() {
51         return x;
52     }
53
54     @Override
55     public int getY() {
56         return y;
57     }
58
59     @Override
60     public String getName() {
61         return name;
62     }
63 }
64 }

```

定义 City 类来对带坐标和名字的顶点建模 (第 39 ~ 63 行)。该程序创建一个拥有 City 类型顶点的图 (第 30 行)。由于 City 实现了 Displayable, 所以为图创建的 GraphView 对象将显示在面板上 (第 33 行)。

作为熟悉图的类和接口的练习, 以合适的边添加一个城市 (例如 Savannah) 到图中。

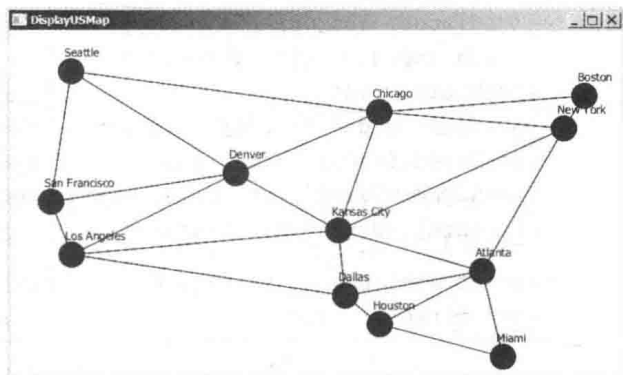


图 28-10 图在面板中显示

复习题

28.9 如果程序清单 28-6 中第 30 ~ 34 行的代码替换为以下代码, 程序清单 28-7 可以工作吗?

```

if (i < v) {
    int x2 = graph.getVertex(v).getX();
    int y2 = graph.getVertex(v).getY();

    // Draw an edge for (i, v)
    getChildren().add(new Line(x1, y1, x2, y2));
}

```

28.10 对于程序清单 28-1 中创建的 graph1 对象, 可以如下创建一个 GraphView 对象吗?

```
GraphView view = new GraphView(graph1);
```

28.6 图的遍历

要点提示: 深度优先和广度优先是遍历图的两个常用方法。

图的遍历 (graph traversal) 是指访问图中的每一个顶点, 且只访问一次的过程。存在两种流行的遍历图的方法: 深度优先遍历 (或深度优先搜索) 和广度优先遍历 (或广度优先搜索)。这两种遍历方法都会产生一个生成树, 它可以用类来建模, 如图 28-11 所示。注意, Tree 是定义在 AbstractGraph 类中的一个内部类。AbstractGraph<V>.Tree 和定义在 25.2.5 节中的 Tree 接口不同。AbstractGraph.Tree 是一个特定的类, 它描述结点的父子关系, 而 Tree 接口定义诸如树的搜索、插入和删除等常用的操作。因为没有必要对生成树执行这些操作, 所以 AbstractGraph<V>.Tree 没有定义为 Tree 的子类型。

在程序清单 28-3 中的第 226 ~ 293 行, 将 Tree 定义为 AbstractGraph 类中的一个内部

类。构造方法给出根、边和搜索顺序来创建一棵树。

`Tree` 类定义了 7 个方法。`getRoot()` 方法返回树的根。可以通过调用 `getSearchOrder()` 方法来获取被搜索顶点的顺序。可以调用 `getParent(v)` 来找出顶点 `v` 在这个搜索中的父结点。调用方法 `getNumberOfVerticesFound()` 返回搜索到的顶点的个数。调用 `getPath(index)` 返回一个从指定下标的顶点到根结点的顶点线性表。调用 `printPath(v)` 显示一条从根结点到顶点 `v` 的路径。可以使用 `printTree()` 方法来显示树中所有的边。

AbstractGraph<V>.Tree	
-root: int	树的根结点
-parent: int[]	结点的父结点
-searchOrder: List<Integer>	遍历结点的顺序
+Tree(root: int, parent: int[], searchOrder: List<Integer>)	给出根、边和搜索顺序来创建一棵树
+getRoot(): int	返回树的根
+getSearchOrder(): List<Integer>	返回被搜索顶点的顺序
+getParent(index: int): int	返回指定下标结点的父结点
+getNumberOfVerticesFound(): int	返回搜索到的顶点的个数
+getPath(index: int): List<V>	返回一个从指定下标的顶点到根结点的顶点线性表
+printPath(index: int): void	显示一条从根结点到指定顶点的路径
+printTree(): void	显示树的根结点和所有的边

图 28-11 `Tree` 类描述具有父子关系的结点

28.7 节和 28.9 节将分别介绍深度优先搜索和广度优先搜索。两种搜索都将产生一个 `Tree` 类的实例。

复习题

28.11 `AbstractGraph<V>.Tree` 实现了程序清单 25-3 中定义的 `Tree` 接口吗？

28.12 使用什么方法来找到树中一个结点的父结点？

28.7 深度优先搜索 (DFS)

要点提示：图的深度优先搜索从图中的一个结点出发，在回溯前尽可能地访问图中的所有结点。

图的深度优先搜索 (DFS) 和 25.2.4 节中讨论的树的深度优先搜索很相似。对于树，搜索从根结点开始；对于图，搜索可以从任意一个顶点开始。

树的深度优先搜索首先访问根结点，然后递归地访问根结点的子树。类似的，图的深度优先搜索首先访问一个顶点，然后递归地访问和这个顶点相连的所有顶点。不同之处在于图可能包含环，这可能会导致无限的递归。为了避免这个问题，需要跟踪已经访问过的顶点。

这种搜索之所以称为深度优先 (depth-first)，是因为它尽可能地搜索图中的“更深处”。搜索从某个顶点 `v` 开始，然后访问顶点 `v` 的第一个未被访问的邻居。如果顶点 `v` 没有未被访问的邻居，返回到达顶点 `v` 的那个顶点。我们假定图是连通的并且从任意结点开始的搜索可以到达所有的结点。如果不是这种情况，参见编程练习题 28.4 以找到图中连通的部分。

28.7.1 DFS 的算法

程序清单 28-8 描述了深度优先搜索算法。

程序清单 28-8 深度优先搜索算法Input: $G = (V, E)$ and a starting vertex v Output: a DFS tree rooted at v

```

1 Tree dfs(vertex v) {
2   visit v;

3   for each neighbor w of v
4     if (w has not been visited) {
5       set v as the parent for w in the tree;
6       dfs(w);
7     }
8 }

```

可以使用一个名为 `isVisited` 的数组，表示一个顶点是否已经被访问过。初始情况下，每个顶点 i 对应的 `isVisited[i]` 都为 `false`。一旦一个顶点 v 被访问过，`isVisited[v]` 就被设置为 `true`。

考虑图 28-12a 中的图。假设从顶点 0 开始深度优先搜索。首先访问 0，然后访问它的任意一个邻居，比如顶点 1。现在，1 已经被访问，如图 28-12b 所示。顶点 1 有三个邻居——顶点 0、2 和 4。由于顶点 0 已经被访问，因而将访问顶点 2 或者顶点 4。选择顶点 2，现在顶点 2 已经被访问，如图 28-12c 所示。顶点 2 有三个邻居，分别为顶点 0、1 和 3。由于顶点 0 和 1 已经被访问，因此选取顶点 3。现在顶点 3 已经被访问，如图 28-12d 所示。此时，顶点已经被以如下的顺序访问：

0, 1, 2, 3

由于顶点 3 的所有邻居都已被访问，因此回溯到顶点 2。由于顶点 2 的所有邻居也都已经被访问，因此回溯到顶点 1。顶点 4 与顶点 1 相连，但是顶点 4 还没有被访问，因此访问顶点 4，如图 28-12e 所示。由于顶点 4 的所有邻居都已被访问，因此回溯到顶点 1。由于顶点 1 的所有邻居都已经被访问，因此返回到顶点 0。由于顶点 0 的所有邻居都已被访问，搜索终止。

由于每条边和每个顶点只被访问一次，所以 `dfs` 方法的时间复杂度为 $O(|E| + |V|)$ ，其中 $|E|$ 表示边的条数， $|V|$ 表示顶点的个数。

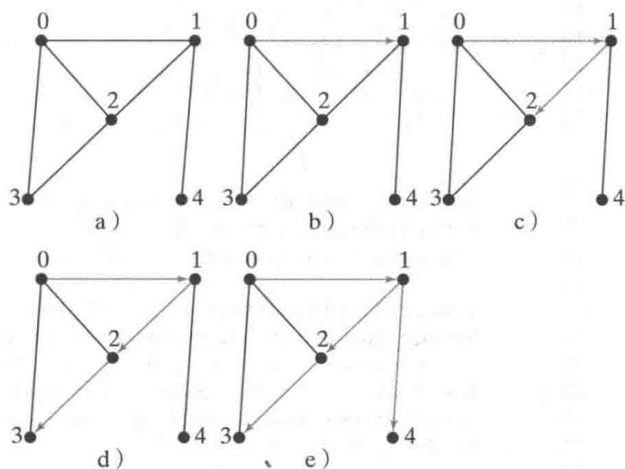


图 28-12 深度优先搜索递归地访问一个结点和它的邻居

28.7.2 DFS 的实现

在程序清单 28-8 中描述的 DFS 算法使用的是递归，很自然地，在实现它的时候也应使用递归。还可以使用栈来实现（参见编程练习题 28.3）。

方法 `dfs(int v)` 在程序清单 28-3 中的第 164 ~ 193 行中实现，它返回一个将顶点 v 作为根结点的 `Tree` 类的实例。该方法将搜索过的顶点存储在一个线性表 `searchOrder` 中（第 165 行），每个顶点的父亲存储在数组 `parent` 中（第 166 行），使用数组 `isVisited` 来表示顶点是否已经被访问过（第 171 行）。调用辅助方法 `dfs(v, parent, searchOrders, isVisited)`

来完成深度优先搜索 (第 174 行)。

在递归的辅助方法中, 搜索从顶点 u 开始。在第 184 行顶点 u 被添加到 `searchOrder` 中, 并且被标记为已访问过 (第 185 行)。对于顶点 u 的每一个未被访问的邻居, 递归地调用方法来执行深度优先搜索。当顶点 $e.v$ 被访问, 顶点 $e.v$ 的父结点将存储在 `parent[e.v]` 中 (第 189 行)。对于一个连通的图或者一个连通的部分, 该方法返回当所有的顶点都被访问的时间。

程序清单 28-9 给出了一个测试程序, 用来显示图 28-1 中由 `chicago` 开始的图的深度优先搜索。由 `chicago` 开始的深度优先搜索的图示如图 28-13 所示。对于 DFS 的交互式 GUI 演示, 参见 www.cs.armstrong.edu/liang/animation/USMapSearch.html。

程序清单 28-9 TestDFS.java

```

1 public class TestDFS {
2     public static void main(String[] args) {
3         String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
4                               "Denver", "Kansas City", "Chicago", "Boston", "New York",
5                               "Atlanta", "Miami", "Dallas", "Houston"};
6
7         int[][] edges = {
8             {0, 1}, {0, 3}, {0, 5},
9             {1, 0}, {1, 2}, {1, 3},
10            {2, 1}, {2, 3}, {2, 4}, {2, 10},
11            {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
12            {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
13            {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
14            {6, 5}, {6, 7},
15            {7, 4}, {7, 5}, {7, 6}, {7, 8},
16            {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
17            {9, 8}, {9, 11},
18            {10, 2}, {10, 4}, {10, 8}, {10, 11},
19            {11, 8}, {11, 9}, {11, 10}
20        };
21
22        Graph<String> graph = new UnweightedGraph<>(vertices, edges);
23        AbstractGraph<String>.Tree dfs =
24            graph.dfs(graph.getIndex("Chicago"));
25
26        java.util.List<Integer> searchOrders = dfs.getSearchOrder();
27        System.out.println(dfs.getNumberOfVerticesFound() +
28            " vertices are searched in this DFS order:");
29        for (int i = 0; i < searchOrders.size(); i++)
30            System.out.print(graph.getVertex(searchOrders.get(i)) + " ");
31        System.out.println();
32
33        for (int i = 0; i < searchOrders.size(); i++)
34            if (dfs.getParent(i) != -1)
35                System.out.println("parent of " + graph.getVertex(i) +
36                    " is " + graph.getVertex(dfs.getParent(i)));
37    }
38 }

```

```

12 vertices are searched in this DFS order:
Chicago Seattle San Francisco Los Angeles Denver
Kansas City New York Boston Atlanta Miami Houston Dallas
parent of Seattle is Chicago
parent of San Francisco is Seattle
parent of Los Angeles is San Francisco
parent of Denver is Los Angeles
parent of Kansas City is Denver
parent of Boston is New York

```

parent of New York is Kansas City
 parent of Atlanta is New York
 parent of Miami is Atlanta
 parent of Dallas is Houston
 parent of Houston is Miami

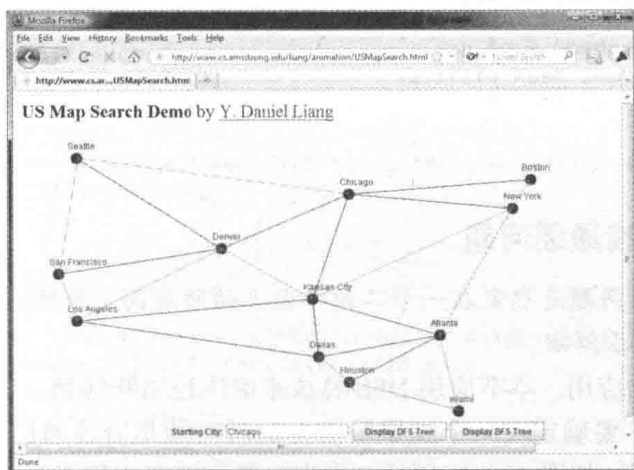


图 28-13 由 Chicago 开始的 DFS 搜索

28.7.3 DFS 的应用

深度优先搜索可以用来解决如下所示的许多问题：

- 检测图是否是连通的。由任何一个顶点开始搜索图，如果搜索的顶点的个数与图中顶点的个数一致，那么图是连通的；否则，图就不是连通的。（参见编程练习题 28.1）。
- 检测两个顶点之间是否存在路径（参见编程练习题 28.5）。
- 找出两个顶点之间的路径（参见编程练习题 28.5）。
- 找出所有连通的部分。一个连通的部分是指一个最大的连通子图，其中每一个顶点到都有路径连接（参见编程练习题 28.4）。
- 检测图中是否存在回路（参见编程练习题 28.6）。
- 找出图中的回路（参见编程练习题 28.7）。
- 找出哈密尔顿路径 / 回路。图中的哈密尔顿路径（Hamiltonian path）是指可以访问图中每个顶点正好一次的路径。哈密尔顿回路（Hamiltonian cycle）是指访问图中每个顶点正好一次并且返回到出发顶点的路径（参见编程练习题 28.17）。

前 6 个问题可以通过使用程序清单 28-3 中的 `dfs` 方法解决。要找到哈密尔顿路径 / 回路，需要探索所有可能的 DFS 来找到具有最长路径的那个。哈密尔顿路径 / 回路具有许多应用，包括解决著名的骑士巡游问题，这个问题在配套网站的补充材料 V.I.E 中给出了。

✓ 复习题

- 28.13 什么是深度优先搜索？
- 28.14 为图 28-3b 中的图从结点 A 开始绘制一个 DFS 树。
- 28.15 为图 28-1 中的图从结点 Atlanta 开始绘制一个 DFS 树。
- 28.16 调用 `dfs(v)` 的返回类型是什么？
- 28.17 程序清单 28-8 中描述的深度优先搜索算法使用了递归。另外，也可以使用栈来实现，如下所示。指出下面算法的错误之处并给出正确的算法。

```
// Wrong version
Tree dfs(vertex v) {
    push v into the stack;
    mark v visited;

    while (the stack is not empty) {
        pop a vertex, say u, from the stack
        visit u;
        for each neighbor w of u
            if (w has not been visited)
                push w into the stack;
    }
}
```

28.8 示例学习：连通圆问题

要点提示：连通圆问题是确定在一个二维平面上的所有圆是否连通的。这个问题可以使用深度优先遍历方法解决。

DFS 算法有许多的应用。本节应用 DFS 算法来解决连通圆问题。

在连通圆问题中，要确定在一个二维平面上的所有圆是否连通的。如果所有圆是连通的，则以填充方式绘制，如图 28-14a 所示。否则，就不填充，如图 28-14b 所示。

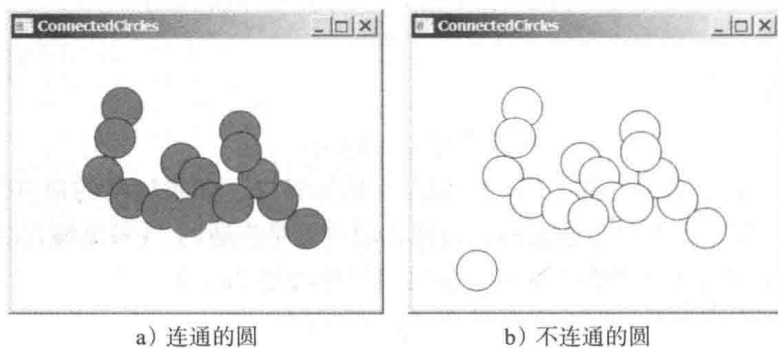


图 28-14 可以使用 DFS 方法来确定是否圆是连通的

我们将编写一个程序，让用户通过在一个没有被圆占据的空白区域点击鼠标来创建一个圆。当圆被添加后，这些圆如果是连通的则被填充绘制，否则不被填充。

将采用图来对问题建模。每个圆是图中的一个顶点。如果两个圆交叉则是连通的。我们在图上应用 DFS，如果深度优先搜索找到了所有的顶点，则图是连通的。

程序在程序清单 28-10 中给出。

程序清单 28-10 ConnectedCircles.java

```
1 import javafx.application.Application;
2 import javafx.geometry.Point2D;
3 import javafx.scene.Node;
4 import javafx.scene.Scene;
5 import javafx.scene.layout.Pane;
6 import javafx.scene.paint.Color;
7 import javafx.scene.shape.Circle;
8 import javafx.stage.Stage;
9
10 public class ConnectedCircles extends Application {
11     @Override // Override the start method in the Application class
12     public void start(Stage primaryStage) {
13         // Create a scene and place it in the stage
```

```

14     Scene scene = new Scene(new CirclePane(), 450, 350);
15     primaryStage.setTitle("ConnectedCircles"); // Set the stage title
16     primaryStage.setScene(scene); // Place the scene in the stage
17     primaryStage.show(); // Display the stage
18 }
19
20 /** Pane for displaying circles */
21 class CirclePane extends Pane {
22     public CirclePane() {
23         this.setOnMouseClicked(e -> {
24             if (!isInsideACircle(new Point2D(e.getX(), e.getY()))) {
25                 // Add a new circle
26                 getChildren().add(new Circle(e.getX(), e.getY(), 20));
27                 colorIfConnected();
28             }
29         });
30     }
31
32     /** Returns true if the point is inside an existing circle */
33     private boolean isInsideACircle(Point2D p) {
34         for (Node circle: this.getChildren())
35             if (circle.contains(p))
36                 return true;
37
38         return false;
39     }
40
41     /** Color all circles if they are connected */
42     private void colorIfConnected() {
43         if (getChildren().size() == 0)
44             return; // No circles in the pane
45
46         // Build the edges
47         java.util.List<AbstractGraph.Edge> edges
48             = new java.util.ArrayList<>();
49         for (int i = 0; i < getChildren().size(); i++)
50             for (int j = i + 1; j < getChildren().size(); j++)
51                 if (overlaps((Circle)getChildren().get(i),
52                     (Circle)getChildren().get(j))) {
53                     edges.add(new AbstractGraph.Edge(i, j));
54                     edges.add(new AbstractGraph.Edge(j, i));
55                 }
56
57         // Create a graph with circles as vertices
58         Graph<Node> graph = new UnweightedGraph<>
59             ((java.util.List<Node>)getChildren(), edges);
60         AbstractGraph<Node>.Tree tree = graph.dfs(0); // a DFS tree
61         boolean isAllCirclesConnected = getChildren().size() == tree
62             .getNumberOfVerticesFound();
63
64         for (Node circle: getChildren()) {
65             if (isAllCirclesConnected) { // All circles are connected
66                 ((Circle)circle).setFill(Color.RED);
67             }
68             else {
69                 ((Circle)circle).setStroke(Color.BLACK);
70                 ((Circle)circle).setFill(Color.WHITE);
71             }
72         }
73     }
74 }
75
76 public static boolean overlaps(Circle circle1, Circle circle2) {
77     return new Point2D(circle1.getCenterX(), circle1.getCenterY()).

```

```

78     distance(circle2.getCenterX(), circle2.getCenterY())
79     <= circle1.getRadius() + circle2.getRadius();
80 }
81 }

```

JavaFX 的 `Circle` 类包含了数据域 `x`、`y` 和 `radius`，给出了圆的圆心位置以及半径。同时还定义了 `contains` 方法来检测一个点是否在圆内。`overlaps` 方法（第 76 ~ 80 行）检测两个圆是否交叉。


当用户在任何已经存在的圆外点击鼠标，一个新的圆在以鼠标点击处为中心的位置创建并添加到面板中（第 26 行）。

为了检测圆是否连通，程序构建了一个图（第 46 ~ 59 行）。圆作为图的顶点。边在第 49 ~ 55 行构建。如果两个圆交叉则代表它们的顶点是连通的（第 51 行）。图的 DFS 结果为一棵树（第 60 行）。树的 `getNumberOfVerticesFound()` 返回查找到的结点数。如果结点数等于圆的个数，则所有的圆是连通的（第 61 ~ 62 行）。

✓ 复习题

- 28.18 解决连通圆问题的图是如何创建的？
- 28.19 当你在圆内点击鼠标时，程序会创建一个新的圆吗？
- 28.20 程序是如何知道所有圆是连通的？

28.9 广度优先搜索 (BFS)

 **要点提示：**图的广度优先搜索逐层访问结点。第一层由起始结点组成，每个下一层由与上一层邻接的结点组成。

图的广度优先遍历与 25.2.4 节中讨论的树的广度优先遍历类似。对于树的广度优先遍历而言，将逐层访问结点。首先访问根结点，然后是根结点的所有孩子，接着是根结点的孙子结点，以此类推。同样的，图的广度优先搜索首先访问一个顶点，然后是所有与其相连的顶点，最后是所有与这些顶点相连的顶点，以此类推。为了确保每个顶点只被访问一次，如果一个顶点已经被访问过，那么就跳过这个顶点。

28.9.1 BFS 的算法

从图中顶点 v 开始的广度优先搜索算法，可以描述为如程序清单 28-11 所示。

程序清单 28-11 广度优先搜索算法

```

1  Tree bfs(vertex v) {
2      create an empty queue for storing vertices to be visited;
3      add v into the queue;
4      mark v visited;
5
6      while (the queue is not empty) {
7          dequeue a vertex, say u, from the queue;
8          add u into a list of traversed vertices;
9          for each neighbor w of u
10             if w has not been visited {
11                 add w into the queue;
12                 set u as the parent for w in the tree;
13                 mark w visited;
14             }
15     }
16 }

```

考虑图 28-15a 中的图。假设从顶点 0 开始广度优先搜索，首先访问顶点 0，然后访问它的所有邻居顶点 1、2、3，如图 28-15b 所示。顶点 1 有三个邻居，顶点 0、2、4。由于顶点 0 和 2 已经被访问，现在只能访问顶点 4，如图 28-15c 所示。顶点 2 有三个邻居，顶点 0、1 和 3，它们都被访问过。顶点 3 有三个邻居，顶点 0、2 和 4，它们也都被访问过。顶点 4 有两个邻居，顶点 1 和 3，它们都被访问过。因此，搜索终止。

由于每条边和每个顶点只访问一次，所以 bfs 方法的时间复杂度为 $O(|E|+|V|)$ ，其中 $|E|$ 表示边的条数， $|V|$ 表示顶点的个数。

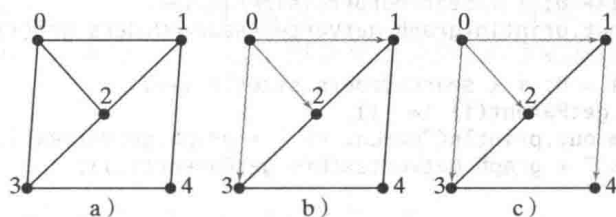


图 28-15 广度优先搜索访问一个结点，然后是其邻居，接着是其邻居的邻居，以此类推

28.9.2 BFS 的实现

方法 `bfs(int v)` 在 `Graph` 接口定义，并且在程序清单 28-3 中的 `AbstractGraph` 类中实现（第 197 ~ 222 行）。它返回一个将顶点 `v` 作为根结点的 `Tree` 类的实例。该方法将搜索到的顶点存储在线性表 `searchOrder` 中（第 198 行），将每个顶点的父结点存储在一个名为 `parent` 的数组中（第 199 行），为队列使用一个链表（第 203 ~ 204 行），使用数组 `isVisited` 来表明顶点是否已经访问过（第 207 行）。这个搜索从顶点 `v` 开始，顶点 `v` 在第 206 行被添加到队列中，并且被标记为已访问（第 207 行）。方法现在检测队列中的每一个顶点 `u`（第 210 行）并且将它添加到 `searchOrder` 中（第 211 行）。方法将顶点 `u` 的每一个未被访问的邻居顶点 `e.v` 添加到队列中（第 214 行），然后设置它的父结点为 `u`（第 215 行），并将其标记为已访问（第 216 行）。

程序清单 28-12 给出了一个测试程序，用来显示图 28-1 中的图从 `chicago` 开始的广度优先搜索。由 `chicago` 开始的广度优先搜索的图示如图 28-16 所示。对于广度优先搜索的交互式的 GUI 演示，参见网址 www.cs.armstrong.edu/liang/animation/USMapSearch.html。

程序清单 28-12 TestBFS.java

```

1 public class TestBFS {
2     public static void main(String[] args) {
3         String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
4                               "Denver", "Kansas City", "Chicago", "Boston", "New York",
5                               "Atlanta", "Miami", "Dallas", "Houston"};
6
7         int[][] edges = {
8             {0, 1}, {0, 3}, {0, 5},
9             {1, 0}, {1, 2}, {1, 3},
10            {2, 1}, {2, 3}, {2, 4}, {2, 10},
11            {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
12            {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
13            {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
14            {6, 5}, {6, 7},
15            {7, 4}, {7, 5}, {7, 6}, {7, 8},
16            {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
17            {9, 8}, {9, 11},

```



```

18     {10, 2}, {10, 4}, {10, 8}, {10, 11},
19     {11, 8}, {11, 9}, {11, 10}
20 };
21
22 Graph<String> graph = new UnweightedGraph<>(vertices, edges);
23 AbstractGraph<String>.Tree bfs =
24     graph.bfs(graph.getIndex("Chicago"));
25
26 java.util.List<Integer> searchOrders = bfs.getSearchOrder();
27 System.out.println(bfs.getNumberOfVerticesFound() +
28     " vertices are searched in this order:");
29 for (int i = 0; i < searchOrders.size(); i++)
30     System.out.println(graph.getVertex(searchOrders.get(i)));
31
32 for (int i = 0; i < searchOrders.size(); i++)
33     if (bfs.getParent(i) != -1)
34         System.out.println("parent of " + graph.getVertex(i) +
35             " is " + graph.getVertex(bfs.getParent(i)));
36 }
37 }

```

12 vertices are searched in this order:

Chicago Seattle Denver Kansas City Boston New York
 San Francisco Los Angeles Atlanta Dallas Miami Houston
 parent of Seattle is Chicago
 parent of San Francisco is Seattle
 parent of Los Angeles is Denver
 parent of Denver is Chicago
 parent of Kansas City is Chicago
 parent of Boston is Chicago
 parent of New York is Chicago
 parent of Atlanta is Kansas City
 parent of Miami is Atlanta
 parent of Dallas is Kansas City
 parent of Houston is Atlanta

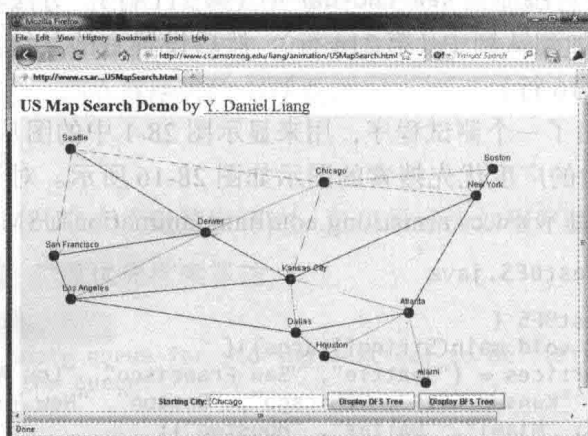


图 28-16 由 chicago 开始的 BFS 搜索

28.9.3 BFS 的应用

许多由深度优先搜索解决的问题也可以由广度优先搜索解决。特别的，广度优先搜索可以用来解决下面的问题：

- 检测图是否是连通的。如果在图中任意两个顶点之间都存在一条路径，那么这个图是连通的。

- 检测在两个顶点之间是否存在路径。
- 找出两个顶点之间的最短路径。可以证明根结点和广度优先搜索树中的任意一个结点之间的路径是根结点和该结点之间的最短路径。(参见复习题 28.25)
- 找出所有的连通部分。一个连通的部分是指一个最大的连通子图，其中的每个顶点对都由路径相连接。
- 检测图中是否存在回路(参见编程练习题 28.6)。
- 找出图中的回路(参见编程练习题 28.7)。
- 检测一个图是否是二分的(如果图的顶点可以分为两个不相交的集合，而且同一个集合中的顶点之间不存在边，那么这个图就是二分的。)(参见编程练习题 28.8)

✓ 复习题

- 28.21 调用 `bfs(v)` 的返回类型是什么?
- 28.22 什么是广度优先搜索?
- 28.23 绘制图 28-3b 中由结点 A 开始的图的广度优先搜索树。
- 28.24 绘制图 28-1 中由结点 Atlanta 开始的图的广度优先搜索树。
- 28.25 证明广度优先搜索树中的根结点和任意结点之间的路径是它们之间的最短路径。

28.10 示例学习：9 枚硬币反面问题

🔑 要点提示：9 枚硬币反面的问题可以简化为最短路径问题。

9 枚硬币反面问题如下：将 9 枚硬币放在一个 3×3 的矩阵中，其中一些正面朝上，另一些正面朝下。一个合法的移动是指翻转任何一个正面朝上的硬币以及与它相邻的硬币(不包括对角线相邻的)。任务就是找到最少次数的移动，使得所有硬币正面朝下。例如，从如图 28-17a 所示的 9 枚硬币开始。当翻动最后一行的第二个硬币之后，9 枚硬币将如图 28-17b 所示。当翻动第一行的第二个硬币之后，9 枚硬币都将正面朝下，如图 28-17c 所示。

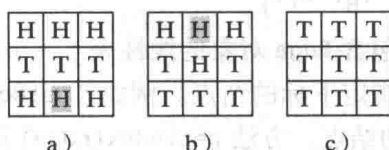


图 28-17 当所有硬币都正面朝下，问题得到解决

下面编写一个程序，提示用户输入 9 枚硬币的一个初始状态，然后显示解决方案，如下面的运行示例所示。

Enter the initial nine coins Hs and Ts: HHHTTTTHHH

The steps to flip the coins are

HHH
TTT
HHH

HHH
THT
TTT

TTT
TTT
TTT

9 枚硬币的每一个状态代表图中的一个结点。例如，图 28-17 中的三个状态对应图中的三个结点。为了方便起见，使用一个 3×3 的矩阵来表示所有的结点，其中 0 表示正面，1 表示背面。由于存在 9 个格子，并且每个格子不是 0 就是 1，因此一共有 2^9 (512) 个结点，分别标记为 0, 1, ..., 511，如图 28-18 所示。

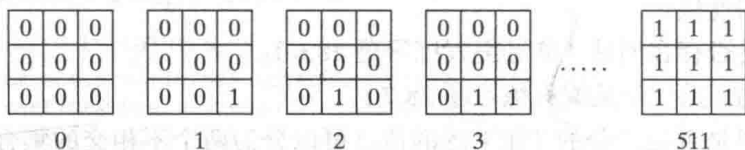


图 28-18 一共有 512 个结点，以 0, 1, 2, ..., 511 的顺序标记

如果存在一个由结点 u 到结点 v 的合法移动，我们就分配一个由结点 v 到结点 u 的边。图 28-19 显示了一个图的部分。注意这里有一条边从 511 到 47，因为可以翻转结点 47 的一个单元格从而成为结点 511。

图 28-18 中的最后一个结点代表 9 枚硬币正面朝下的状态。为方便起见，我们称最后一个结点为目标结点 (target node)，这样，目标结点被标记为 511。假设 9 枚硬币反面的问题的初始状态对应到结点 s ，那么问题就简化为搜索结点 s 和目标结点之间的最短路径，这就等价于在一个以目标结点为根结点的广度优先搜索树中搜索从结点 s 到目标结点的路径。

现在的任务是创建一个标记为 0, 1, 2, ..., 511 的包含 512 个结点的图，并且顶点之间有边相连。一旦图被创建，就得到以结点 511 为根结点的一个广度优先搜索树。从这个广度优先搜索树，可以找到从根结点到任意一个结点的最短路径。创建一个名为 `NineTailModel` 的类，其中包含了获取从目标结点到任意其他结点之间最短路径的方法。这个类的 UML 图如图 28-20 所示。

结点被可视化表示为一个包含字母 H 和 T 的 3×3 的矩阵。在程序中，我们使用一个包含 9 个字符的一维数组来表示一个结点。例如，图 28-18 中的顶点 1 的结点在数组中表示为 {'H', 'H', 'H', 'H', 'H', 'H', 'H', 'H', 'T'}。

方法 `getEdges()` 返回一个包含 `Edge` 对象的线性表。

方法 `getNode(index)` 返回指定下标的结点。例如，`getNode(0)` 返回包含 9 个 H 的结点，`getNode(511)` 返回包含 9 个 T 的结点。方法 `getIndex(node)` 返回结点的下标。

注意，数据域 `tree` 定义为保护的，因此它们可以被下一章中的子类 `WeightedNineTail` 访问。

方法 `getFlippedNode(char[] node, int position)` 翻转指定位置和其邻接位置的结点。这个方法返回新结点的下标。位置是从 0 到 8 的一个值，指向了结点中的一个硬币，如下图所示。



例如，图 28-19 中的结点 56，在位置 0 处翻转，那么将会得到结点 51。如果在位置 1 处翻转结点 56，将会得到结点 47。

方法 `flipACell(char[] node, int row, int column)` 翻转指定行和列的结点。例如，如果在第 0 行第 0 列翻转结点 56，那么新结点为 408。如果在第 2 行第 0 列翻转结点 56，那么新结点为 30。

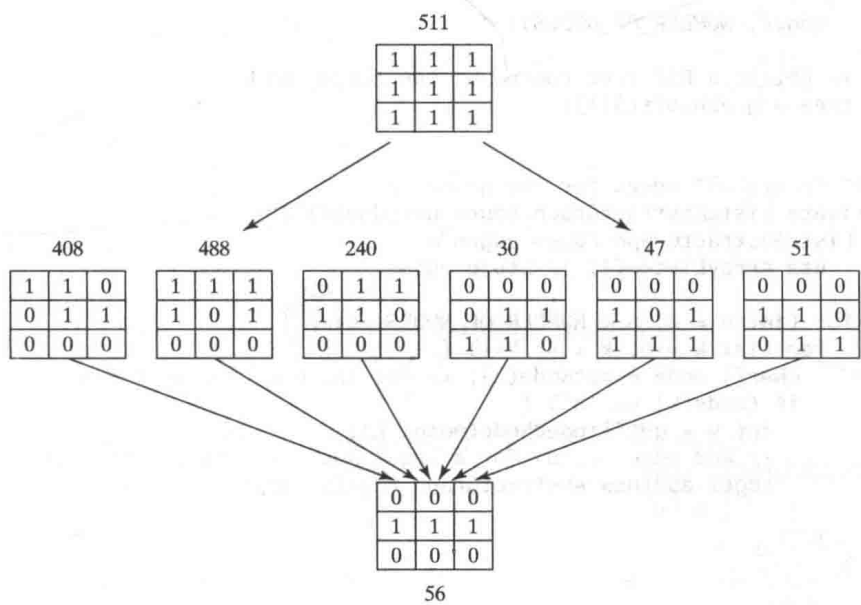


图 28-19 如果在格子翻转后，结点 u 变为结点 v，就分配一个由结点 v 到结点 u 的边

NineTailModel	
#tree: AbstractGraph<Integer>.Tree	一棵根结点位于 511 的树
+NineTailModel()	为 9 枚硬币反面问题构建一个模型并获得该树
+getShortestPath(nodeIndex: int): List<Integer>	返回一个从指定结点到根结点的路径。返回的路径由线性表中的结点标签组成
-getEdges(): List<AbstractGraph.Edge>	为图返回一个 Edge 对象的线性表
+getNode(index: int): char[]	返回一个由 9 个 H 和 T 字符组成的结点
+getIndex(node: char[]): int	返回指定结点的下标
+getFlippedNode(node: char[], position: int): int	翻转结点指定位置的硬币以及它的相邻位置硬币，返回被翻转的结点的下标
+flipACell(node: char[], row: int, column: int): void	翻转结点的指定行和列位置硬币
+printNode(node: char[]): void	在控制台打印结点

图 28-20 NineTailModel 类使用一个图建模 9 枚硬币反面的问题

程序清单 28-13 给出了 NineTailModel.java 的源代码。

程序清单 28-13 NineTailModel.java

```
1 import java.util.*;
2
3 public class NineTailModel {
4     public final static int NUMBER_OF_NODES = 512;
5     protected AbstractGraph<Integer>.Tree tree; // Define a tree
6
7     /** Construct a model */
8     public NineTailModel() {
9         // Create edges
10        List<AbstractGraph.Edge> edges = getEdges();
11
12        // Create a graph
13        UnweightedGraph<Integer> graph = new UnweightedGraph<>(
```

```

14     edges, NUMBER_OF_NODES);
15
16     // Obtain a BSF tree rooted at the target node
17     tree = graph.bfs(511);
18 }
19
20 /** Create all edges for the graph */
21 private List<AbstractGraph.Edge> getEdges() {
22     List<AbstractGraph.Edge> edges =
23         new ArrayList<>(); // Store edges
24
25     for (int u = 0; u < NUMBER_OF_NODES; u++) {
26         for (int k = 0; k < 9; k++) {
27             char[] node = getNode(u); // Get the node for vertex u
28             if (node[k] == 'H') {
29                 int v = getFlippedNode(node, k);
30                 // Add edge (v, u) for a legal move from node u to node v
31                 edges.add(new AbstractGraph.Edge(v, u));
32             }
33         }
34     }
35
36     return edges;
37 }
38
39 public static int getFlippedNode(char[] node, int position) {
40     int row = position / 3;
41     int column = position % 3;
42
43     flipACell(node, row, column);
44     flipACell(node, row - 1, column);
45     flipACell(node, row + 1, column);
46     flipACell(node, row, column - 1);
47     flipACell(node, row, column + 1);
48
49     return getIndex(node);
50 }
51
52 public static void flipACell(char[] node, int row, int column) {
53     if (row >= 0 && row <= 2 && column >= 0 && column <= 2) {
54         // Within the boundary
55         if (node[row * 3 + column] == 'H')
56             node[row * 3 + column] = 'T'; // Flip from H to T
57         else
58             node[row * 3 + column] = 'H'; // Flip from T to H
59     }
60 }
61
62 public static int getIndex(char[] node) {
63     int result = 0;
64
65     for (int i = 0; i < 9; i++)
66         if (node[i] == 'T')
67             result = result * 2 + 1;
68         else
69             result = result * 2 + 0;
70
71     return result;
72 }
73
74 public static char[] getNode(int index) {
75     char[] result = new char[9];
76
77     for (int i = 0; i < 9; i++) {

```

For example:
 index: 3
 node: HHHHHHTT

H	H	H
H	H	H
H	T	T

For example:
 node: THHHHHHTT
 index: 259

```

78     int digit = index % 2;
79     if (digit == 0)
80         result[8 - i] = 'H';
81     else
82         result[8 - i] = 'T';
83     index = index / 2;
84 }
85
86 return result;
87 }
88
89 public List<Integer> getShortestPath(int nodeIndex) {
90     return tree.getPath(nodeIndex);
91 }
92
93 public static void printNode(char[] node) {
94     for (int i = 0; i < 9; i++)
95         if (i % 3 != 2)
96             System.out.print(node[i]);
97         else
98             System.out.println(node[i]);
99     System.out.println();
100 }
101 }
102 }

```

For example:
node: THHHHHHTT
Output:

T	H	H
H	H	H
H	T	T

构造方法 (第 8 ~ 18 行) 创建一个有 512 个结点的图, 其中每一条边对应着从一个结点到另一个结点的移动 (第 10 行)。从这个图, 可以得到一个以目标结点 511 为根结点的广度优先搜索树 (第 17 行)。

为了创建边, 方法 `getEdges` (第 21 ~ 37 行) 检测每一个结点 u , 查看它是否可以翻转成为另一个结点 v 。如果可以, 将 (v, u) 添加到 `Edge` 线性表 (第 31 行)。方法 `getFlippedNode(node, position)` 通过在一个结点中翻转一个 H 格子和它的邻居来找到翻转的结点 (第 43 ~ 47 行)。方法 `flipACell (node, row, column)` 真正在一个结点中翻转一个 H 格子和它的邻居 (第 52 ~ 60 行)。

方法 `getIndex(node)` 实现的方式与将二进制数转换为十进制数的方式相似 (第 62 ~ 72 行)。方法 `getNode(index)` 返回一个包含字母 H 和 T 的结点 (第 74 ~ 87 行)。

方法 `getShortestpath(nodeIndex)` 调用方法 `getPath(nodeIndex)` 来获取从指定的结点到目标结点之间的最短路径的顶点 (第 89 ~ 91 行)。

方法 `printNode(node)` 在控制台上显示一个结点 (第 93 ~ 101 行)。

程序清单 28-14 给出一个程序, 提示用户输入一个初始结点, 并且显示到达目标结点的步骤。

程序清单 28-14 NineTail.java

```

1  import java.util.Scanner;
2
3  public class NineTail {
4      public static void main(String[] args) {
5          // Prompt the user to enter nine coins' Hs and Ts
6          System.out.print("Enter the initial nine coins Hs and Ts: ");
7          Scanner input = new Scanner(System.in);
8          String s = input.nextLine();
9          char[] initialNode = s.toCharArray();
10
11         NineTailModel model = new NineTailModel();
12         java.util.List<Integer> path =

```

```

13         model.getShortestPath(NineTailModel.getIndex(initialNode));
14
15     System.out.println("The steps to flip the coins are ");
16     for (int i = 0; i < path.size(); i++)
17         NineTailModel.printNode(
18             NineTailModel.getNode(path.get(i).intValue()));
19 }
20 }

```

该程序提示用户输入一个包含 9 个 H 和 T 字母的初始结点，就像第 8 行中的字符串，从字符串中得到一个字符数组（第 9 行），用一个模型来创建图并得到广度优先搜索树（第 11 行），得到一个从初始结点到目标结点的最短路径（第 12 ~ 13 行），然后显示这个路径上的结点（第 16 ~ 18 行）。

复习题

- 28.26 NineTailModel 中图的结点是如何创建的？
- 28.27 NineTailModel 中图的边是如何创建的？
- 28.28 在程序清单 28-13 中调用 `getIndex("HTHTTTTHHH".toCharArray())` 会返回什么？在程序清单 28-13 中调用 `getNode(46)` 会返回什么？
- 28.29 程序清单 28-13 中第 26 行和第 27 行交换，程序还会工作吗？为什么？

关键术语

adjacency list (邻接线性表)	incident edges (连接边)
adjacency matrix (邻接矩阵)	parallel edge (平行边)
adjacent vertices (邻接顶点)	Seven Bridges of Königsberg (哥尼斯堡七孔桥问题)
breadth-first search (广度优先搜索)	simple graph (简单图)
complete graph (完全图)	spanning tree (生成树)
cycle (回路)	tree (树)
degree (度)	undirected graph (无向图)
depth-first search (深度优先搜索)	unweighted graph (非加权图)
directed graph (有向图)	weighted graph (加权图)
graph (图)	

本章小结

1. 图是一种有用的数学结构，可以表示现实世界中实体之间的联系。已经学习了如何使用类和接口来对图建模，如何使用数组和链表来表示顶点和边，以及如何实现图的操作。
2. 图的遍历是指访问图中的每个顶点一次并且只有一次的过程。学习了两种遍历图的常用方法：深度优先搜索 (DFS) 和广度优先搜索 (BFS)。
3. 深度优先搜索和广度优先搜索可以解决许多问题，如检测图是否连通，检测图中是否存在环，找出两个顶点之间最短路径等。

测试题

回答位于网址 www.cs.armstrong.edu/liang/intro10e/quiz.html 的本章测试题。

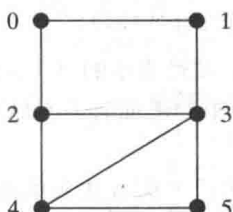
编程练习题

28.6 ~ 28.10 节

- *28.1 (检测一个图是否是连通的) 编写一个程序，它从文件读入图并且判定该图是否是连通的。文件

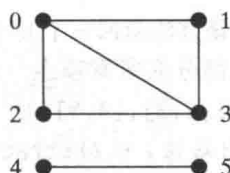
中的第一行包含了表明顶点个数的数字 (n)。顶点被标记为 $0, 1, \dots, n-1$ 。接下来的每一行, 以 $u \ v_1 \ v_2 \dots$ 的形式描述边 (u, v_1) 、 (u, v_2) , 以此类推。图 28-21 给出了两个文件对应的图的例子。

```
File
6
0 1 2
1 0 3
2 0 3 4
3 1 2 4 5
4 2 3 5
5 3 4
```



a)

```
File
6
0 1 2 3
1 0 3
2 0 3
3 0 1 2
4 5
5 4
```



b)

图 28-21 图的顶点和边存储在一个文件中

程序应该提示用户输入文件的名字, 应该从文件中读取数据, 创建 `UnweightedGraph` 的一个实例 `g`, 然后调用 `g.printEdges()` 来显示所有的边, 并调用 `dfs()` 来获取 `AbstractGraph.Tree` 的一个实例 `tree`。如果 `tree.getNumberOfVerticeFound()` 与图中的顶点数目相同, 那么图就是连通的。下面是这个程序的运行示例:

```
Enter a file name: c:\exercise\GraphSample1.txt Enter
The number of vertices is 6
Vertex 0: (0, 1) (0, 2)
Vertex 1: (1, 0) (1, 3)
Vertex 2: (2, 0) (2, 3) (2, 4)
Vertex 3: (3, 1) (3, 2) (3, 4) (3, 5)
Vertex 4: (4, 2) (4, 3) (4, 5)
Vertex 5: (5, 3) (5, 4)
The graph is connected
```

提示: 使用 `new UnweightedGraph(list, numberOfVertices)` 来创建一个图, 其中 `list` 是包含 `AbstractGraph.Edge` 对象的一个线性表。使用 `new AbstractGraph.Edge(u, v)` 来创建一条边。读取第一行来获取顶点的数目。将接下来的每一行读入一个字符串 `s` 中并且使用 `s.split("[\\s+])` 来从字符串中提取顶点并产生从顶点开始的边。

*28.2 (为图创建文件) 修改程序清单 28-1 来创建一个文件 `graph1`。文件形式在编程练习题 28.1 中描述。从程序清单 28-1 中第 8 ~ 21 行定义的数组创建这个文件。图的顶点数为 12, 它存储在文件的第一行。文件的内容应该如下所示:

```
12
0 1 3 5
1 0 2 3
2 1 3 4 10
3 0 1 2 4 5
4 2 3 5 7 8 10
5 0 3 4 6 7
6 5 7
7 4 5 6 8
8 4 7 9 10 11
9 8 11
10 2 4 8 11
11 8 9 10
```

*28.3 (使用堆栈实现深度优先搜索) 程序清单 28-8 描述的深度优先搜索使用的是递归。设计一个新的算法而不使用递归实现它。使用伪代码描述该算法。通过定义一个名为 `UnweightedGraphWithNonrecursiveDFS` 的新类来实现它, 该类继承自 `UnweightedGraph` 并且覆盖 `dfs` 方法。

- *28.4 (寻找连通部分) 创建一个名为 `MyGraph` 的新类作为 `UnweightedGraph` 的子类, 其中包含找到图中所有连通部分的方法, 方法头如下:

```
public List<List<Integer>> getConnectedComponents();
```

该方法返回一个 `List<List<Integer>>`。线性表中的每个元素是另一个线性表, 它包含了连通部分的所有顶点。例如, 对于图 28-21b 中的图而言, `getConnectedComponents()` 返回 `[[0,1,2,3],[4,5]]`。

- *28.5 (找出路径) 在 `AbstractGraph` 中添加一个使用下面方法头的新方法, 找出两个顶点之间的路径:

```
public List<Integer> getPath(int u, int v);
```

该方法会返回一个 `List<Integer>`, 它包含由顶点 `u` 到顶点 `v` 的路径上的所有顶点。使用广度优先搜索方法, 可以获取由顶点 `u` 到顶点 `v` 的最短路径。如果顶点 `u` 和顶点 `v` 之间不存在路径, 方法返回 `null`。

- *28.6 (探测回路) 在 `AbstractGraph` 中添加一个使用下面方法头的新方法, 判定图中是否存在环:

```
public boolean isCyclic();
```

- *28.7 (找出回路) 在 `AbstractGraph` 添加一个使用下面方法头的新方法, 找出图中的环:

```
public List<Integer> getACycle(int u);
```

该方法返回一个 `List`, 它包含从顶点 `u` 开始的回路上的所有顶点。如果图中没有回路, 方法返回 `null`。

- **28.8 (测试二分图) 回顾一下, 如果图的顶点可以分为两个不相交的集合, 而且同一个集合中的顶点之间不存在边, 那么这个图是二分的。在 `AbstractGraph` 中添加一个新方法来检测图是否是二分的:

```
public boolean isBipartite();
```

- **28.9 (得到二分集合) 在 `AbstractGraph` 中添加一个新方法, 如果图是二分的, 返回这两个二分集合:

```
public List<List<Integer>> getBipartite();
```

该方法返回一个包含两个子线性表的 `List`, 每一个都包含了一个顶点集合。如果图不是二分的, 方法返回 `null`。

- 28.10 (找出最短路径) 编写一个程序, 从文件中读取一个连通图。图存储在一个文件中, 使用和编程练习题 28.1 中指定的一样的格式。程序应该提示用户输入文件名, 然后输入两个顶点, 最后显示两个顶点之间的最短路径。例如, 对于图 28-21a 中的图, 顶点 0 和顶点 5 之间的最短路径可以显示为 0 1 3 5。

下面是该程序的一个运行示例:

```
Enter a file name: c:\exercise\GraphSample1.txt Enter
Enter two vertices (integer indexes): 0 5 Enter
The number of vertices is 6
Vertex 0: (0, 1) (0, 2)
Vertex 1: (1, 0) (1, 3)
Vertex 2: (2, 0) (2, 3) (2, 4)
Vertex 3: (3, 1) (3, 2) (3, 4) (3, 5)
Vertex 4: (4, 2) (4, 3) (4, 5)
Vertex 5: (5, 3) (5, 4)
The path is 0 1 3 5
```

- **28.11** (修改程序清单 28-14) 程序清单 28-14 中的程序允许用户在控制台上为 9 枚硬币反面问题输入数据并且在控制台上显示结果。编写一个程序, 让用户设置 9 枚硬币的初始状态 (如图 28-22a 所示), 然后单击 Solve 按钮来显示解决方案, 如图 28-22b 所示。初始情况下, 用户可以通过单击鼠标来翻转硬币。将翻转的单元设置为红色。



图 28-22 解决 9 枚硬币反面问题的程序

- **28.12** (9 枚硬币反面问题的变体) 在 9 枚硬币反面问题中, 当翻转一个正面的硬币时, 水平和垂直方向上的邻居也都被翻转。重新编写程序, 假设对角线上的邻居也都被翻转。
- **28.13** (4×4 16 枚硬币反面问题) 程序清单 28-14, 提供了 9 枚硬币反面问题的解答。修改该程序, 成为一个 4×4 的矩阵中放置了 16 枚硬币。注意可能对于一个开始的模式并不存在解答。如果是这样, 报告没有解答存在。
- **28.14** (4×4 16 枚硬币反面问题的分析) 本书中的 9 枚硬币反面问题使用的是 3×3 的矩阵。假设在一个 4×4 的矩阵中放置了 16 枚硬币。编写一个程序, 找出不存在解答的开始模式的数目。
- *28.15** (4×4 16 枚硬币反面问题的 GUI) 修改编程练习题 28.14, 使得用户可以设置 4×4 16 枚硬币反面问题的初始化模式 (参见图 28-23a)。用户可以单击 Solve 按钮来显示解答, 如图 28-23b 所示。开始时, 用户可以点击鼠标按钮来翻转硬币。如果解答不存在, 显示一条信息来报告该消息。

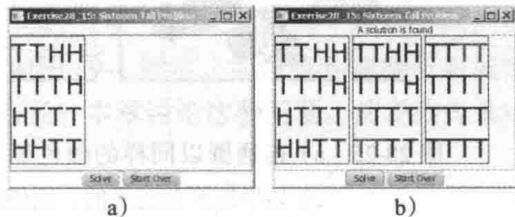


图 28-23 解决 16 枚硬币反面问题的程序

- **28.16** (诱导子图) 给定一个无向图 $G=(V, E)$ 和一个整数 k , 找出 G 的一个最大的诱导子图 H , H 中的所有结点的度 $\geq k$, 或者得到这样的子图不存在的结论。使用下面的文件头实现这个方法:

```
public static Graph maxInducedSubgraph(Graph g, int k)
```

如果这样的子图不存在, 方法返回 null。

{ } 提示: 一个直观的方法是删除那些度小于 k 的顶点。随着顶点及其邻接边被删除, 其他顶点的度可能会减小。继续这个过程直到没有顶点被删除, 或者所有的顶点都被删除。

- ***28.17** (哈密尔顿环) 补充材料 VI.E 给出了哈密尔顿路径算法的实现。在 Graph 接口中添加以下 getHamiltonianCycle 方法, 并且在 AbstractGraph 类中实现它:

```
/** Return a Hamiltonian cycle
 * Return null if the graph doesn't contain a Hamiltonian cycle */
public List<Integer> getHamiltonianCycle()
```

- ***28.18** (骑士巡游回路) 改写补充材料 VI.E 中示例学习的 KnightTourApp.java 程序, 找出骑士访问棋盘的每个方块并且返回到起始方块的路径。将骑士巡游回路问题简化为寻找哈密尔顿环的问题。

- **28.19** (显示一个图中的深度优先搜索 / 广度优先搜索树) 修改程序清单 28-6 中的 GraphView, 添加一个数据域 tree 和一个 set 方法。树中的边显示为红色。编写一个程序, 显示图 28-1 中的图, 以及从一个指定城市出发的深度优先搜索 / 广度优先搜索树, 如图 28-13 和图 28-16 所示。如果输入了一个地图中没有的城市, 程序在一个标签中给出错误信息。

***28.20 (显示图)** 编写一个程序，从一个文件中读取一个图，然后显示它。文件的第一行包含了表示顶点个数的数字 (n)。顶点被标记为 $0, 1, \dots, n-1$ 。接下来的每一行，用 $u \ x \ y \ v1 \ v2$ 的格式描述了 u 的位置 (x, y) 以及边 $(u, v1)$ 、 $(u, v2)$ ，以此类推。图 28-24a 给出了对应图的文件例子。程序提示用户输入文件名，从文件中读取数据并且使用 `GraphView` 在面板上显示图，如图 28-24b 所示。

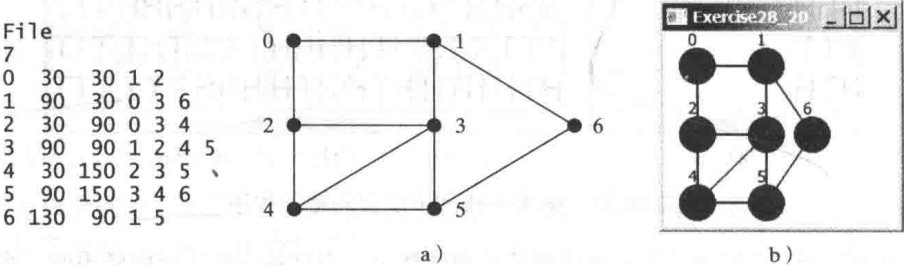


图 28-24 程序读取关于图的信息并且可视化显示它

****28.21 (显示连通圆集合)** 修改程序清单 28-10，以不同颜色显示连通圆的集合。也就是说，如果两个圆是连通的，则使用相同的颜色显示。否则，它们的颜色不同，如图 28-25 所示。(提示：参见编程练习题 28.4。)

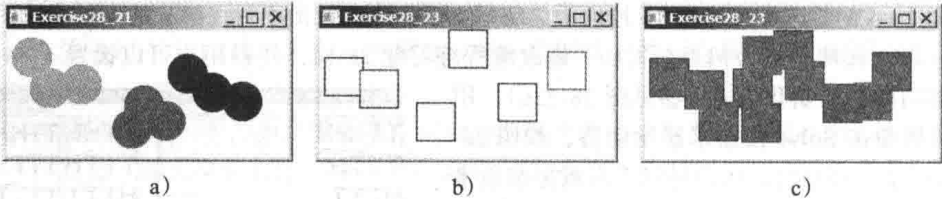


图 28-25 a) 连通圆以同样的颜色显示；b) 如果矩形不是连通的，则不使用颜色填充；c) 如果矩形是连通的，则使用颜色填充

- **28.22 (移动圆)** 修改程序清单 28-10，使得用户可以拖放和移动圆。
- **28.23 (连通矩形)** 程序清单 28-10 允许用户创建圆并确定它们是否是连通的。为矩形重写该程序。程序使得用户可以在没有被矩形占据的空白区域点击鼠标来创建矩形。当矩形被添加，如果一些矩形是连通的则以填充方式绘制，否则不填充。如图 28-25b ~ 图 28-25c 所示。
- *28.24 (删除圆)** 修改程序清单 28-10，使得用户可以通过在圆内点击删除圆。

加权图及其应用

教学目标

- 使用邻接矩阵和邻接线性表来表示加权边 (29.2 节)。
- 使用扩展自 `AbstractGraph` 类的 `WeightedGraph` 类来建模加权图 (29.3 节)。
- 设计并实现得到最小生成树的算法 (29.4 节)。
- 通过扩展 `Tree` 类来定义 `MST` 类 (29.4 节)。
- 设计并实现算法, 找出单源最短路径 (29.5 节)。
- 定义继承自 `Tree` 类的 `ShortestPathTree` 类 (29.5 节)。
- 用最短路径算法来解决加权九枚硬币反面的问题 (29.6 节)。

29.1 引言

要点提示: 如果图的每条边都赋予一个权重, 则该图是一个加权图。加权图有很多实际的应用。

图 28-1 假设图表示了城市之间的飞行次数。可以应用 BFS 来找到两个城市之间的最小飞行次数。假设边代表了城市之间的驾驶距离, 如图 29-1 所示。如何找到连接所有城市的最小总距离呢? 又如何找到两个城市之间的最小路径? 本章讨论这些问题。前者称为最小生成树 (MST) 问题, 后者是最短路径问题。

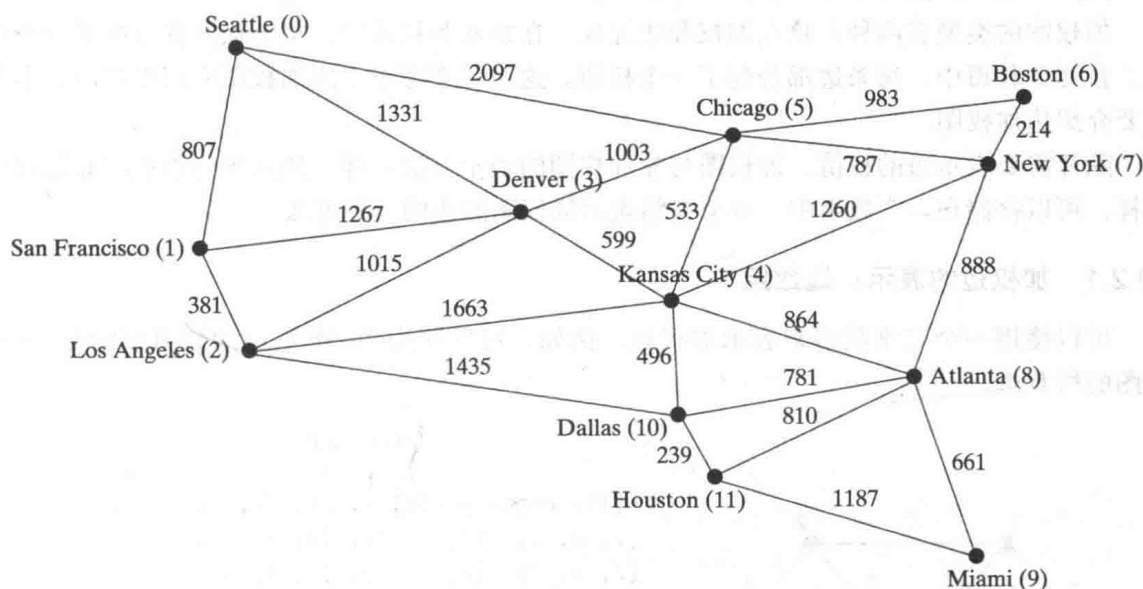


图 29-1 该图对城市之间的距离进行了建模

前面章节中介绍了图的概念。你学会了如何使用边数组、边线性表、邻接矩阵和邻接线性表来表示边, 以及如何使用 `Graph` 接口、`AbstractGraph` 类和 `UnweightedGraph` 类来对图

建模。前面章节中还介绍了两种重要的遍历图的方法：深度优先搜索和广度优先搜索，并将其应用于解决实际问题。本章将介绍加权图。29.4 节介绍找出最小生成树的算法，29.5 节介绍找出最短路径的算法。

教学注意：在开始介绍加权图的算法和应用之前，通过网址 www.cs.armstrong.edu/liang/animation/WeightedGraphLearningTool.html 提供的交互式工具来了解下加权图是很有帮助的，如图 29-2 所示。该工具可以让你输入顶点，指定边以及它们的权重，查看图，从一个单一源中找到一个 MST 以及所有 shortest 路径，如图 29-2 所示。

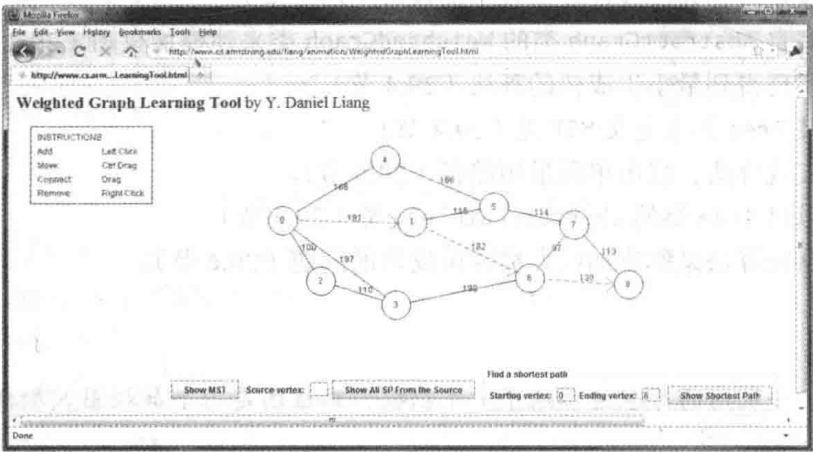


图 29-2 可以使用该工具，通过鼠标操作来创建一个加权图，显示 MST 和最短路径

29.2 加权图的表示

要点提示：加权边可以存储在邻接线性表中。

加权图的类型有两种：顶点加权和边加权。在顶点加权图中，每个顶点都分配了一个权值。在边加权图中，每条边都分配了一个权值。这两种类型中，边加权图应用更广泛，本章主要介绍边加权图。

除开需要表示边的权值，加权图与非加权图的表示方法一样。加权图的顶点与非加权图一样，可以存储在一个数组中。本节介绍表示加权图的边的三种方法。

29.2.1 加权边的表示：边数组

可以使用一个二维数组来表示加权边。例如，可以使用 29-3b 所示的数组存储图 29-3a 中图的所有边。

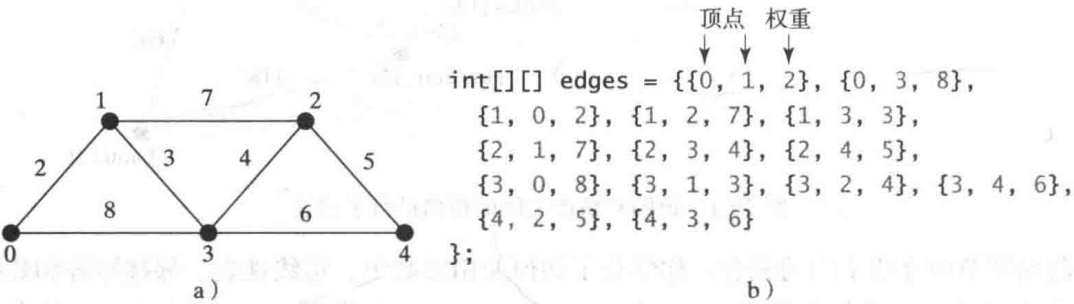


图 29-3 加权图中每条边都分配了一个权重

{ } 注意：权值可以为任何类型：Integer, Double, BigDecimal, 等等。可以采用一个如下所示的 Object 类型的二维数组：

```
Object[][] edges = {
    {new Integer(0), new Integer(1), new SomeTypeForWeight(2)},
    {new Integer(0), new Integer(3), new SomeTypeForWeight(8)},
    ...
};
```

29.2.2 加权邻接矩阵

假设图有 n 个顶点。可以使用一个 $n \times n$ 的二维矩阵 `weights` 来表示边上的权值。`weights[i][j]` 表示边 (i, j) 上的权值。如果顶点 i 和 j 不相连, `weights[i][j]` 为 `null`。例如, 在图 29-3a 中图的权值可以使用邻接矩阵表示, 如下所示:

<pre>Integer[][] adjacencyMatrix = { {null, 2, null, 8, null}, {2, null, 7, 3, null}, {null, 7, null, 4, 5}, {8, 3, 4, null, 6}, {null, null, 5, 6, null} };</pre>	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th></th> <th>0</th> <th>1</th> <th>2</th> <th>3</th> <th>4</th> </tr> </thead> <tbody> <tr> <th>0</th> <td>null</td> <td>2</td> <td>null</td> <td>8</td> <td>null</td> </tr> <tr> <th>1</th> <td>2</td> <td>null</td> <td>7</td> <td>3</td> <td>null</td> </tr> <tr> <th>2</th> <td>null</td> <td>7</td> <td>null</td> <td>4</td> <td>5</td> </tr> <tr> <th>3</th> <td>8</td> <td>3</td> <td>4</td> <td>null</td> <td>6</td> </tr> <tr> <th>4</th> <td>null</td> <td>null</td> <td>5</td> <td>6</td> <td>null</td> </tr> </tbody> </table>		0	1	2	3	4	0	null	2	null	8	null	1	2	null	7	3	null	2	null	7	null	4	5	3	8	3	4	null	6	4	null	null	5	6	null
	0	1	2	3	4																																
0	null	2	null	8	null																																
1	2	null	7	3	null																																
2	null	7	null	4	5																																
3	8	3	4	null	6																																
4	null	null	5	6	null																																

29.2.3 邻接线性表

另一种表示边的方法是将边定义为对象。程序清单 28-3 中 `AbstractGraph.Edge` 类用来表示非加权的边。对于加权图, 我们定义如程序清单 29-1 所示的 `WeightedEdge` 类。

程序清单 29-1 `WeightedEdge.java`

```
1 public class WeightedEdge extends AbstractGraph.Edge
2     implements Comparable<WeightedEdge> {
3     public double weight; // The weight on edge (u, v)
4
5     /** Create a weighted edge on (u, v) */
6     public WeightedEdge(int u, int v, double weight) {
7         super(u, v);
8         this.weight = weight;
9     }
10
11     @Override /** Compare two edges on weights */
12     public int compareTo(WeightedEdge edge) {
13         if (weight > edge.weight)
14             return 1;
15         else if (weight == edge.weight)
16             return 0;
17         else
18             return -1;
19     }
20 }
```

`AbstractGraph.Edge` 是一个定义在 `AbstractGraph` 类中的内部类。它表示一条由顶点 u 到顶点 v 的边。`WeightedEdge` 继承自 `AbstractGraph.Edge`, 添加了一个新的属性 `weight`。

为了创建一个 `WeightedEdge` 对象, 使用 `new WeightedEdge(i, j, w)`, 其中 w 是边 (i, j) 上的权值。通常你会需要比较边的权重。因此, `WeightedEdge` 类实现了 `Comparable` 接口。

对于非加权图, 我们使用邻接链表来表示边。对于加权图, 我们依然使用邻接线性表, 图 29-3a 中图的顶点的邻接线性表可以表示为:


```
java.util.List<WeightedEdge>[] list = new java.util.List[5];
```

list[0]	WeightedEdge(0, 1, 2)	WeightedEdge(0, 3, 8)		
list[1]	WeightedEdge(1, 0, 2)	WeightedEdge(1, 2, 3)	WeightedEdge(1, 2, 7)	
list[2]	WeightedEdge(2, 3, 4)	WeightedEdge(2, 4, 5)	WeightedEdge(2, 1, 7)	
list[3]	WeightedEdge(3, 1, 3)	WeightedEdge(3, 2, 4)	WeightedEdge(3, 4, 6)	WeightedEdge(3, 0, 8)
list[4]	WeightedEdge(4, 2, 5)	WeightedEdge(4, 3, 6)		

list[i] 存储所有与顶点 i 相邻的边。

为了灵活性，我们使用一个数组线性表而不是固定大小的数组来表示 list，如下所示：

```
List<List<WeightedEdge>> list = new java.util.ArrayList<>();
```

复习题

- 29.1 对于代码 `WeightedEdge edge = new WeightedEdge(1, 2, 3, 5)` 而言，`edge.u`、`edge.v` 以及 `edge.weight` 是什么？
- 29.2 下面代码的输出是什么？

```
List<WeightedEdge> list = new ArrayList<>();
list.add(new WeightedEdge(1, 2, 3.5));
list.add(new WeightedEdge(2, 3, 4.5));
WeightedEdge e = java.util.Collections.max(list);
System.out.println(e.u);
System.out.println(e.v);
System.out.println(e.weight);
```

29.3 WeightedGraph 类

要点提示：WeightedGraph 类继承自 AbstractGraph。

前面章节设计了 Graph 接口、AbstractGraph 类和 UnweightedGraph 类来对图建模。遵从这一模式，我们设计 AbstractGraph 的子类 WeightedGraph，如图 29-4 所示。

WeightedGraph 简单地继承自 AbstractGraph，采用 5 个构造方法来创建具体 WeightedGraph 实例。WeightedGraph 继承了 AbstractGraph 的所有方法，重写了 Clear 和 addVertex 方法，并且实现了新的方法 addEdge 以添加一个加权边，同时还引入了新的方法来获得最小生成树并找出单源的所有最短路径。最小生成树和最短路径将分别在 29.4 节和 29.5 节中介绍。

程序清单 29-2 实现了 WeightedGraph。内部使用边的邻接线性表（第 38 ~ 63 行）来存储每个顶点的邻接边。每当创建一个 WeightedGraph，就会产生它的边的邻接线性表（第 47 和 57 行）。方法 getMinimumSpanningTree()（第 99 ~ 138 行）和 getShortestPaths()（第 156 ~ 197 行）将在后面小节中介绍。

程序清单 29-2 WeightedGraph.java

```
1 import java.util.*;
2
3 public class WeightedGraph<V> extends AbstractGraph<V> {
4     /** Construct an empty */
5     public WeightedGraph() {
6     }
7
8     /** Construct a WeightedGraph from vertices and edged in arrays */
9     public WeightedGraph(V[] vertices, int[][] edges) {
10         createWeightedGraph(java.util.Arrays.asList(vertices), edges);
```



图 29-4 WeightedGraph 继承自 AbstractGraph

```

11 }
12
13 /** Construct a WeightedGraph from vertices and edges in list */
14 public WeightedGraph(int[][] edges, int numberOfVertices) {
15     List<V> vertices = new ArrayList<>();
16     for (int i = 0; i < numberOfVertices; i++)
17         vertices.add((V)(new Integer(i)));
18
19     createWeightedGraph(vertices, edges);
20 }
21
22 /** Construct a WeightedGraph for vertices 0, 1, 2 and edge list */
23 public WeightedGraph(List<V> vertices, List<WeightedEdge> edges) {
24     createWeightedGraph(vertices, edges);
25 }
26
27 /** Construct a WeightedGraph from vertices 0, 1, and edge array */
28 public WeightedGraph(List<WeightedEdge> edges,
29     int numberOfVertices) {
30     List<V> vertices = new ArrayList<>();
31     for (int i = 0; i < numberOfVertices; i++)
32         vertices.add((V)(new Integer(i)));
33
34     createWeightedGraph(vertices, edges);
35 }
36
37 /** Create adjacency lists from edge arrays */

```

```

38 private void createWeightedGraph(List<V> vertices, int[][] edges) {
39     this.vertices = vertices;
40
41     for (int i = 0; i < vertices.size(); i++) {
42         neighbors.add(new ArrayList<Edge>()); // Create a list for vertices
43     }
44
45     for (int i = 0; i < edges.length; i++) {
46         neighbors.get(edges[i][0]).add(
47             new WeightedEdge(edges[i][0], edges[i][1], edges[i][2]));
48     }
49 }
50
51 /** Create adjacency lists from edge lists */
52 private void createWeightedGraph(
53     List<V> vertices, List<WeightedEdge> edges) {
54     this.vertices = vertices;
55
56     for (int i = 0; i < vertices.size(); i++) {
57         neighbors.add(new ArrayList<Edge>()); // Create a list for vertices
58     }
59
60     for (WeightedEdge edge: edges) {
61         neighbors.get(edge.u).add(edge); // Add an edge into the list
62     }
63 }
64
65 /** Return the weight on the edge (u, v) */
66 public double getWeight(int u, int v) throws Exception {
67     for (Edge edge : neighbors.get(u)) {
68         if (edge.v == v) {
69             return ((WeightedEdge)edge).weight;
70         }
71     }
72
73     throw new Exception("Edge does not exist");
74 }
75
76 /** Display edges with weights */
77 public void printWeightedEdges() {
78     for (int i = 0; i < getSize(); i++) {
79         System.out.print(getVertex(i) + " (" + i + "): ");
80         for (Edge edge : neighbors.get(i)) {
81             System.out.print("(" + edge.u +
82                 ", " + edge.v + ", " + ((WeightedEdge)edge).weight + ") ");
83         }
84         System.out.println();
85     }
86 }
87
88 /** Add edges to the weighted graph */
89 public boolean addEdge(int u, int v, double weight) {
90     return addEdge(new WeightedEdge(u, v, weight));
91 }
92
93 /** Get a minimum spanning tree rooted at vertex 0 */
94 public MST getMinimumSpanningTree() {
95     return getMinimumSpanningTree(0);
96 }
97
98 /** Get a minimum spanning tree rooted at a specified vertex */
99 public MST getMinimumSpanningTree(int startingVertex) {
100     // cost[v] stores the cost by adding v to the tree
101     double[] cost = new double[getSize()];

```

```

102     for (int i = 0; i < cost.length; i++) {
103         cost[i] = Double.POSITIVE_INFINITY; // Initial cost
104     }
105     cost[startingVertex] = 0; // Cost of source is 0
106
107     int[] parent = new int[getSize()]; // Parent of a vertex
108     parent[startingVertex] = -1; // startingVertex is the root
109     double totalWeight = 0; // Total weight of the tree thus far
110
111     List<Integer> T = new ArrayList<>();
112
113     // Expand T
114     while (T.size() < getSize()) {
115         // Find smallest cost v in V - T
116         int u = -1; // Vertex to be determined
117         double currentMinCost = Double.POSITIVE_INFINITY;
118         for (int i = 0; i < getSize(); i++) {
119             if (!T.contains(i) && cost[i] < currentMinCost) {
120                 currentMinCost = cost[i];
121                 u = i;
122             }
123         }
124
125         T.add(u); // Add a new vertex to T
126         totalWeight += cost[u]; // Add cost[u] to the tree
127
128         // Adjust cost[v] for v that is adjacent to u and v in V - T
129         for (Edge e : neighbors.get(u)) {
130             if (!T.contains(e.v) && cost[e.v] > ((WeightedEdge)e).weight) {
131                 cost[e.v] = ((WeightedEdge)e).weight;
132                 parent[e.v] = u;
133             }
134         }
135     } // End of while
136
137     return new MST(startingVertex, parent, T, totalWeight);
138 }
139
140 /** MST is an inner class in WeightedGraph */
141 public class MST extends Tree {
142     private double totalWeight; // Total weight of all edges in the tree
143
144     public MST(int root, int[] parent, List<Integer> searchOrder,
145         double totalWeight) {
146         super(root, parent, searchOrder);
147         this.totalWeight = totalWeight;
148     }
149
150     public double getTotalWeight() {
151         return totalWeight;
152     }
153 }
154
155 /** Find single source shortest paths */
156 public ShortestPathTree getShortestPath(int sourceVertex) {
157     // cost[v] stores the cost of the path from v to the source
158     double[] cost = newdouble[getSize()];
159     for (int i = 0; i < cost.length; i++) {
160         cost[i] = Double.POSITIVE_INFINITY; // Initial cost set to infinity
161     }
162     cost[sourceVertex] = 0; // Cost of source is 0
163
164     // parent[v] stores the previous vertex of v in the path
165     int[] parent = newint[getSize()];

```

```

166     parent[sourceVertex] = -1; // The parent of source is set to -1
167
168     // T stores the vertices whose path found so far
169     List<Integer> T = new ArrayList<>();
170
171     // Expand T
172     while (T.size() < getSize()) {
173         // Find smallest cost v in V - T
174         int u = -1; // Vertex to be determined
175         double currentMinCost = Double.POSITIVE_INFINITY;
176         for (int i = 0; i < getSize(); i++) {
177             if (!T.contains(i) && cost[i] < currentMinCost) {
178                 currentMinCost = cost[i];
179                 u = i;
180             }
181         }
182
183         T.add(u); // Add a new vertex to T
184
185         // Adjust cost[v] for v that is adjacent to u and v in V - T
186         for (Edge e : neighbors.get(u)) {
187             if (!T.contains(e.v)
188                 && cost[e.v] > cost[u] + ((WeightedEdge)e).weight) {
189                 cost[e.v] = cost[u] + ((WeightedEdge)e).weight;
190                 parent[e.v] = u;
191             }
192         }
193     } // End of while
194
195     // Create a ShortestPathTree
196     return new ShortestPathTree(sourceVertex, parent, T, cost);
197 }
198
199 /** ShortestPathTree is an inner class in WeightedGraph */
200 public class ShortestPathTree extends Tree {
201     private double[] cost; // cost[v] is the cost from v to source
202
203     /** Construct a path */
204     public ShortestPathTree(int source, int[] parent,
205         List<Integer> searchOrder, double[] cost) {
206         super(source, parent, searchOrder);
207         this.cost = cost;
208     }
209
210     /** Return the cost for a path from the root to vertex v */
211     public double getCost(int v) {
212         return cost[v];
213     }
214
215     /** Print paths from all vertices to the source */
216     public void printAllPaths() {
217         System.out.println("All shortest paths from " +
218             vertices.get(getRoot()) + " are:");
219         for (int i = 0; i < cost.length; i++) {
220             printPath(i); // Print a path from i to the source
221             System.out.println("(cost: " + cost[i] + ")"); // Path cost
222         }
223     }
224 }
225 }

```

WeightedGraph 类继承自 AbstractGraph (第 3 行)。AbstractGraph 中的属性 vertices 和 neighbors 被 WeightedGraph 继承。neighbors 是一个线性表。线性表中的每个元素是包含了边的另一个线性表。对于无权图来说, 每条边是 AbstractGraph.Edge 的一个实例。对

于加权图来说, 每条边是 `WeightedEdge` 的一个实例。`WeightedEdge` 是 `Edge` 的一个子类型。因此, 对于加权图来说, 可以添加一个加权边到 `neighbors.get(i)` 中 (第 47 行)。

程序清单 29-3 给出了一个测试程序, 为图 29-1 所表示的创建一个图以及为图 29-3a 所表示的创建另外一个图。

程序清单 29-3 TestWeightedGraph.java

```

1 public class TestWeightedGraph {
2     public static void main(String[] args) {
3         String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
4             "Denver", "Kansas City", "Chicago", "Boston", "New York",
5             "Atlanta", "Miami", "Dallas", "Houston"};
6
7         int[][] edges = {
8             {0, 1, 807}, {0, 3, 1331}, {0, 5, 2097},
9             {1, 0, 807}, {1, 2, 381}, {1, 3, 1267},
10            {2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
11            {3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599},
12            {3, 5, 1003},
13            {4, 2, 1663}, {4, 3, 599}, {4, 5, 533}, {4, 7, 1260},
14            {4, 8, 864}, {4, 10, 496},
15            {5, 0, 2097}, {5, 3, 1003}, {5, 4, 533},
16            {5, 6, 983}, {5, 7, 787},
17            {6, 5, 983}, {6, 7, 214},
18            {7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},
19            {8, 4, 864}, {8, 7, 888}, {8, 9, 661},
20            {8, 10, 781}, {8, 11, 810},
21            {9, 8, 661}, {9, 11, 1187},
22            {10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},
23            {11, 8, 810}, {11, 9, 1187}, {11, 10, 239}
24        };
25
26        WeightedGraph<String> graph1 =
27            new WeightedGraph<>(vertices, edges);
28        System.out.println("The number of vertices in graph1: "
29            + graph1.getSize());
30        System.out.println("The vertex with index 1 is "
31            + graph1.getVertex(1));
32        System.out.println("The index for Miami is " +
33            graph1.getIndex("Miami"));
34        System.out.println("The edges for graph1:");
35        graph1.printWeightedEdges();
36
37        edges = new int[][] {
38            {0, 1, 2}, {0, 3, 8},
39            {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
40            {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
41            {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
42            {4, 2, 5}, {4, 3, 6}
43        };
44        WeightedGraph<Integer> graph2 = new WeightedGraph<>(edges, 5);
45        System.out.println("\nThe edges for graph2:");
46        graph2.printWeightedEdges();
47    }
48 }

```

```

The number of vertices in graph1: 12
The vertex with index 1 is San Francisco
The index for Miami is 9
The edges for graph1:
Vertex 0: (0, 1, 807) (0, 3, 1331) (0, 5, 2097)
Vertex 1: (1, 2, 381) (1, 0, 807) (1, 3, 1267)

```

```

Vertex 2: (2, 1, 381) (2, 3, 1015) (2, 4, 1663) (2, 10, 1435)
Vertex 3: (3, 4, 599) (3, 5, 1003) (3, 1, 1267)
        (3, 0, 1331) (3, 2, 1015)
Vertex 4: (4, 10, 496) (4, 8, 864) (4, 5, 533) (4, 2, 1663)
        (4, 7, 1260) (4, 3, 599)
Vertex 5: (5, 4, 533) (5, 7, 787) (5, 3, 1003)
        (5, 0, 2097) (5, 6, 983)
Vertex 6: (6, 7, 214) (6, 5, 983)
Vertex 7: (7, 6, 214) (7, 8, 888) (7, 5, 787) (7, 4, 1260)
Vertex 8: (8, 9, 661) (8, 10, 781) (8, 4, 864)
        (8, 7, 888) (8, 11, 810)
Vertex 9: (9, 8, 661) (9, 11, 1187)
Vertex 10: (10, 11, 239) (10, 4, 496) (10, 8, 781) (10, 2, 1435)
Vertex 11: (11, 10, 239) (11, 9, 1187) (11, 8, 810)

The edges for graph2:
Vertex 0: (0, 1, 2) (0, 3, 8)
Vertex 1: (1, 0, 2) (1, 2, 7) (1, 3, 3)
Vertex 2: (2, 3, 4) (2, 1, 7) (2, 4, 5)
Vertex 3: (3, 1, 3) (3, 4, 6) (3, 2, 4) (3, 0, 8)
Vertex 4: (4, 2, 5) (4, 3, 6)

```

程序在第 3 ~ 27 行为图 29-1 中的图创建 graph1。第 3 ~ 5 行定义 graph1 的顶点。第 7 ~ 24 行定义 graph1 的边。边采用二维数组表示。对于数组中的每一行 i, edges[i][0] 和 edges[i][1] 表示存在一个由顶点 edges[i][0] 到顶点 edges[i][1] 的边, 并且这条边的权值为 edges[i][2]。例如, {0,1,807} (第 8 行) 表示由顶点 0(edges[0][0]) 到顶点 1(edges[0][1]) 的边, 并且权重为 807(edges[0][2])。{0,5,2097} (第 8 行) 表示由顶点 0(edges[2][0]) 到顶点 5(edges[2][1]) 的边, 并且权重为 2097(edges[2][2])。第 35 行调用 graph1 中的方法 printWeightedEdges() 来显示 graph1 中的所有边。

程序在第 37 ~ 44 行为图 29-3a 中的图 graph2 创建边。第 46 行调用 graph2 中的方法 printWeightedEdges() 来显示 graph2 中的所有边。

复习题

29.3 如果使用优先队列来存储加权边, 下面代码的输出是什么?

```

PriorityQueue<WeightedEdge> q = new PriorityQueue<>();
q.offer(new WeightedEdge(1, 2, 3.5));
q.offer(new WeightedEdge(1, 6, 6.5));
q.offer(new WeightedEdge(1, 7, 1.5));
System.out.println(q.poll().weight);
System.out.println(q.poll().weight);
System.out.println(q.poll().weight);

```

29.4 如果使用优先队列来存储加权边, 下面代码中有什么错误? 修改这些错误并显示输出。

```

List<PriorityQueue<WeightedEdge>> queues = new ArrayList<>();
queues.get(0).offer(new WeightedEdge(0, 2, 3.5));
queues.get(0).offer(new WeightedEdge(0, 6, 6.5));
queues.get(0).offer(new WeightedEdge(0, 7, 1.5));
queues.get(1).offer(new WeightedEdge(1, 0, 3.5));
queues.get(1).offer(new WeightedEdge(1, 5, 8.5));
queues.get(1).offer(new WeightedEdge(1, 8, 19.5));
System.out.println(queues.get(0).peek()
    .compareTo(queues.get(1).peek()));

```

29.4 最小生成树

 **要点提示:** 图的最小生成树是一个具有最小的总权重的生成树。

一个图可能有很多生成树。假设边具有权值, 最小生成树拥有最小的权值和。例如,

图 29-5b、图 29-5c、图 29-5d 中的树都是图 29-5a 中图的生成树。图 29-3c 和图 29-3d 中的树是最小生成树。

找到最小生成树的问题有很多应用。考虑一个在很多城市拥有分公司的公司，公司想要架设电话线来连接所有的分公司。电话公司对不同城市之间的连接要价不同。存在很多种方法可以将所有分公司连接在一起，最便宜的方法就是找出一棵费用总和最小的生成树。

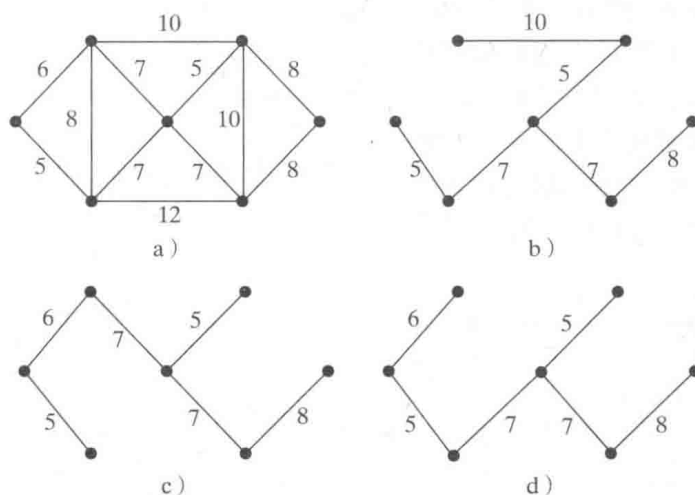


图 29-5 c 和 d 中的树都是 a 中图的最小生成树

29.4.1 最小生成树算法

如何找出最小生成树？对于这个问题，有几个著名的算法。本节将介绍 Prim 算法。Prim 算法从包含任意顶点的生成树 T 开始。算法通过反复添加与已经在树中的顶点相连的具有最短边的顶点来对树进行扩展。Prim 算法是一种贪婪算法，其描述在程序清单 29-4 中。

程序清单 29-4 Prim 的最小生成树算法

```

1  MST minimumSpanningTree() {
2    Let T be a set for the vertices in the spanning tree;
3    Initially, add the starting vertex to T;
4
5    while (size of T < n) {
6      Find u in T and v in V - T with the smallest weight
7      on the edge (u, v), as shown in Figure 29.6;
8      Add v to T and set parent[v] = u;
9    }
10 }
```

算法从将起始顶点添加到 T 中开始，然后持续地将顶点（比方说 v ）从 $V-T$ 添加到 T 中。 v 是与 T 中顶点相邻的顶点中边权值最小的那个顶点。例如，存在 5 条边连接着 T 和 $V-T$ 之间的顶点，如图 29-6 所示，其中 (u, v) 就是权值最小的那个。考虑图 29-7 中的图。算法以如下的顺序将顶点添加到 T 中：

1) 将顶点 0 添加到 T 中。

2) 将顶点 5 添加到 T 中，因为 $\text{Edge}(5, 0, 5)$ 在所有与 T 中的顶点相连的边中拥有最小的权值，如图 29-7a 所示。从 0 指向 5 的箭头表示 0 是 5 的父顶点。

3) 将顶点 1 添加到 T 中，因为 $\text{Edge}(1, 0, 6)$ 在所有与 T 中的顶点相连的边中拥有最小的权值，如图 29-7b 所示。

4) 将顶点 6 添加到 T 中, 因为 $\text{Edge}(6, 1, 7)$ 在所有与 T 中的顶点相连的边中拥有最小的权值, 如图 29-7c 所示。

5) 将顶点 2 添加到 T 中, 因为 $\text{Edge}(2, 6, 5)$ 在所有与 T 中的顶点相连的边中拥有最小的权值, 如图 29-7d 所示。

6) 将顶点 4 添加到 T 中, 因为 $\text{Edge}(4, 6, 7)$ 在所有与 T 中的顶点相连的边中拥有最小的权值, 如图 29-7e 所示。

7) 将顶点 3 添加到 T 中, 因为 $\text{Edge}(3, 2, 8)$ 在所有与 T 中的顶点相连的边中拥有最小的权值, 如图 29-7f 所示。

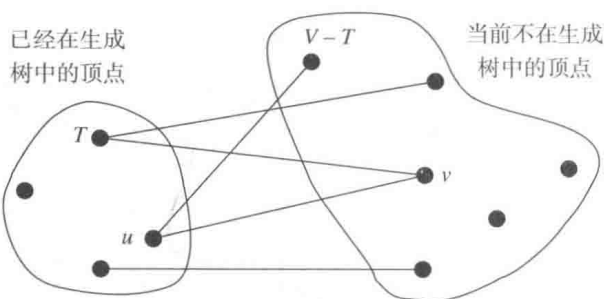


图 29-6 找到 T 中的顶点 u , 该顶点连接 $V-T$ 中的顶点 v 具有最小权值

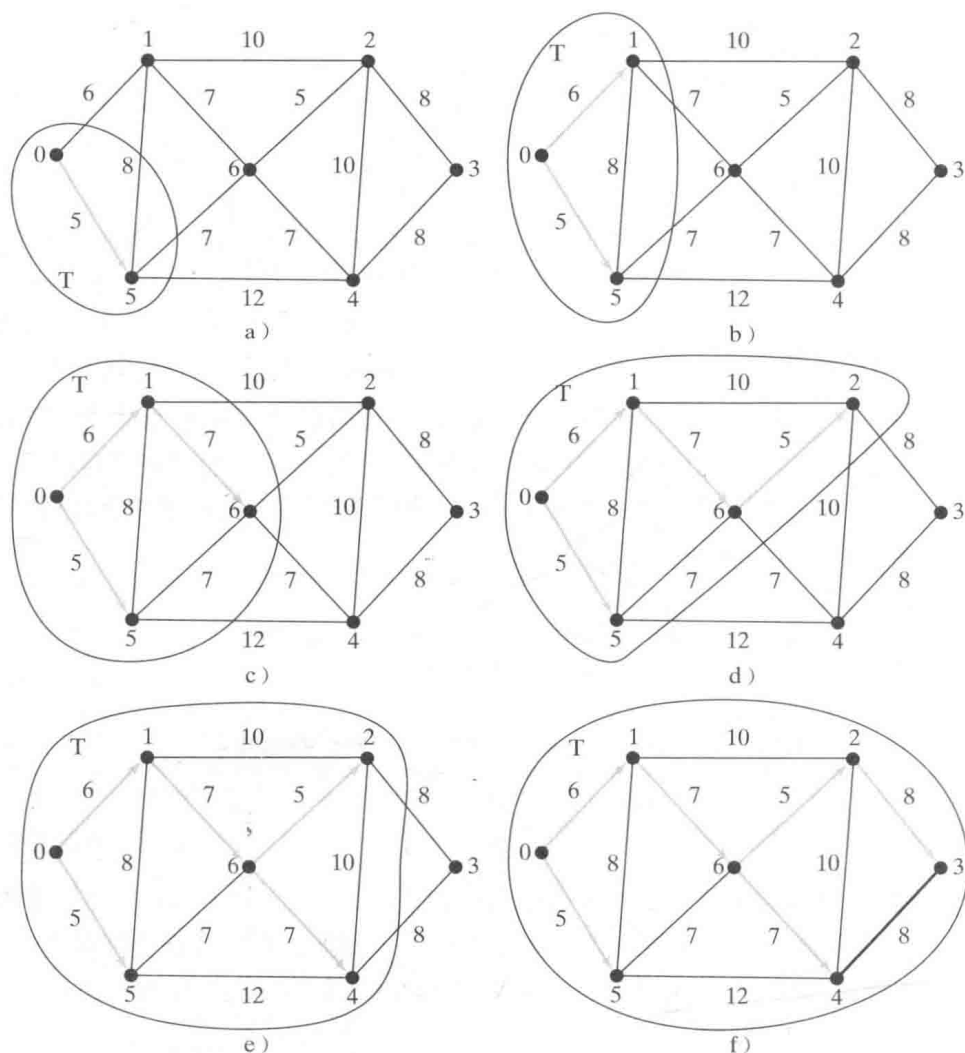


图 29-7 具有最小权值的邻接顶点被不断地添加到 T 中

注意: 最小生成树不是唯一的。例如, 图 29-5 中的 c 和 d 都是图 29-5a 中图的最小生成树。然而, 如果权值是不同的, 那么图就只有唯一的最小生成树。

注意: 这里假设图是连通且无向的。如果一个图不是连通的或者有向的, 这里的算法是

无效的。可以修改算法，为任何无向图找出生成森林。生成森林是一种图，该图中每个连通的组件是一棵树。

29.4.2 完善 Prim 的 MST 算法

为了容易地确定加入到树中的下一个顶点，使用 `cost[v]` 存储加入顶点 `v` 到生成树 `T` 中的开销。初始的，对于起始顶点来说 `cost[s]` 为 0，而对于其他顶点设置 `cost[v]` 值为无穷大。算法重复地在 `V-T` 中找到具有最小 `cost[u]` 顶点 `u`，并将其移到 `T` 中。完善后的算法在程序清单 29-5 中给出。

程序清单 29-5 完善后的 Prim 算法

```
1 MST getMinimumSpanngingTree(s) {
2   Let T be a set that contains the vertices in the spanning tree;
3   Initially T is empty;
4   Set cost[s] = 0; and cost[v] = infinity for all other vertices in V;
5
6   while (size of T < n) {
7     Find u not in T with the smallest cost[u];
8     Add u to T;
9     for (each v not in T and (u, v) in E)
10      if (cost[v] > w(u, v)) { // Adjust cost[v]
11        cost[v] = w(u, v); parent[v] = u;
12      }
13   }
14 }
```

29.4.3 MST 算法的实现

方法 `getMinimumSpanningTree(int v)` 定义在 `WeightedGraph` 类中，它返回一个 `MST` 类的实例，参见图 29-4。`MST` 类定义为继承自 `Tree` 类的 `WeightedGraph` 类中的一个内部类，如图 29-8 所示。`Tree` 类在图 28-11 中给出。`MST` 类在程序清单 29-2 中的第 141 ~ 153 行实现。

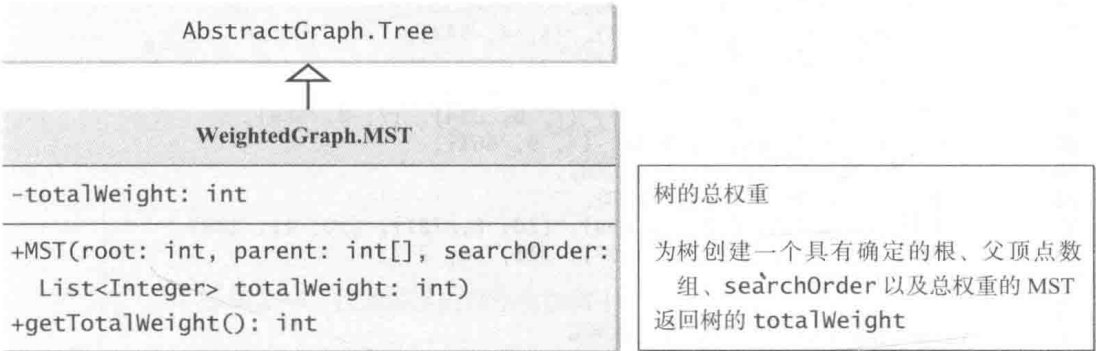


图 29-8 MST 类继承自 Tree 类

完善后的 Prim 算法大大简化了实现。方法 `getMinimumSpanningTree` 使用了改善后的 Prim 算法实现，在程序清单 29-2 中的第 99 ~ 138 行中。方法 `getMinimumSpanningTree(int startingVertex)` 设置 `cost[startingVertex]` 为 0 (第 105 行)，为其他顶点设置 `cost[v]` 为无穷大 (第 102 ~ 104 行)。`startingVertex` 的父顶点设为 -1 (第 108 行)。`T` 是存储添加到生成树的顶点的线性表 (第 111 行)。使用线性表而不是集合来表示 `T` 是因为用于记录加入到 `T` 的顶点的次序。

初始的，`T` 为空。为了扩充 `T`，方法执行以下操作：

1) 找到具有最小 $\text{cost}[u]$ 的顶点 u (第 118 ~ 123 行) 并且将其加入到 T 中 (第 125 行)。
 2) 添加 u 到 T 中后, 如果 $\text{cost}[v] > w(u, v)$, 则对 $V-T$ 中的顶点 u 的每个邻接顶点 v 更新 $\text{cost}[v]$ 和 $\text{parent}[v]$ (第 129 ~ 134 行)。

在一个新顶点被添加到 T 中之后, 更新 totalWeight (第 126 行)。一旦所有的顶点都被添加到 T 中, 就创建一个 MST 的实例 (第 137 行)。注意, 如果图不是连通的, 那么这个方法就不起作用。然而, 可以修改它来获得一个局部的 MST。

MST 类扩展 *Tree* 类 (第 141 行)。为了创建一个 MST 的实例, 传递 root 、 parent 、 T 和 totalWeight (第 144 ~ 145 行)。数据域 root 、 parent 和 searchOrder 定义在 *Tree* 类中, *Tree* 类是定义在 *AbstractGraph* 中的一个内部类。

注意, 因为 T 是一个线性表, 通过调用 $T.\text{contains}(i)$ 检测顶点 i 是否在 T 中需要 $O(n)$ 的时间。因此, 这个实现的总体时间复杂度为 $O(n^3)$ 。有兴趣的读者可以参考编程练习题 29.20 来改善实现, 缩减复杂度为 $O(n^2)$ 。

程序清单 29-6 给出了一个测试程序, 分别显示图 29-1 中的图的最小生成树和图 29-3a 中的图。

程序清单 29-6 TestMinimumSpanningTree.java

```

1 public class TestMinimumSpanningTree {
2     public static void main(String[] args) {
3         String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
4             "Denver", "Kansas City", "Chicago", "Boston", "New York",
5             "Atlanta", "Miami", "Dallas", "Houston"};
6
7         int[][] edges = {
8             {0, 1, 807}, {0, 3, 1331}, {0, 5, 2097},
9             {1, 0, 807}, {1, 2, 381}, {1, 3, 1267},
10            {2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
11            {3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599},
12            {3, 5, 1003},
13            {4, 2, 1663}, {4, 3, 599}, {4, 5, 533}, {4, 7, 1260},
14            {4, 8, 864}, {4, 10, 496},
15            {5, 0, 2097}, {5, 3, 1003}, {5, 4, 533},
16            {5, 6, 983}, {5, 7, 787},
17            {6, 5, 983}, {6, 7, 214},
18            {7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},
19            {8, 4, 864}, {8, 7, 888}, {8, 9, 661},
20            {8, 10, 781}, {8, 11, 810},
21            {9, 8, 661}, {9, 11, 1187},
22            {10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},
23            {11, 8, 810}, {11, 9, 1187}, {11, 10, 239}
24        };
25
26        WeightedGraph<String> graph1 =
27            new WeightedGraph<>(vertices, edges);
28        WeightedGraph<String> MST tree1 = graph1.getMinimumSpanningTree();
29        System.out.println("Total weight is " + tree1.getTotalWeight());
30        tree1.printTree();
31
32        edges = new int[][]{
33            {0, 1, 2}, {0, 3, 8},
34            {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
35            {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
36            {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
37            {4, 2, 5}, {4, 3, 6}
38        };
39
40        WeightedGraph<Integer> graph2 = new WeightedGraph<>(edges, 5);

```

```

41     WeightedGraph<Integer>.MST tree2 =
42         graph2.getMinimumSpanningTree(1);
43     System.out.println("\nTotal weight is " + tree2.getTotalWeight());
44     tree2.printTree();
45 }
46 }
    
```

```

Total weight is 6513.0
Root is: Seattle
Edges: (Seattle, San Francisco) (San Francisco, Los Angeles)
       (Los Angeles, Denver) (Denver, Kansas City) (Kansas City, Chicago)
       (New York, Boston) (Chicago, New York) (Dallas, Atlanta)
       (Atlanta, Miami) (Kansas City, Dallas) (Dallas, Houston)

Total weight is 14.0
Root is: 1
Edges: (1, 0) (3, 2) (1, 3) (2, 4)
    
```

程序在第 27 行为图 29-1 创建一个加权图，接着调用 `getMinimumSpanningTree()` (第 28 行) 来返回一个表示图最小生成树的 MST。调用 MST 对象的 `printTree()` (第 30 行) 显示树中的边。注意，MST 是 `Tree` 类的子类。方法 `printTree()` 定义在 `Tree` 类中。

最小生成树的图示如图 29-9 所示。顶点以如下的顺序添加到树中：西雅图、圣弗朗西斯科、洛杉矶、丹佛、堪萨斯城、达拉斯、休斯敦、芝加哥、纽约、波士顿、亚特兰大和迈阿密。

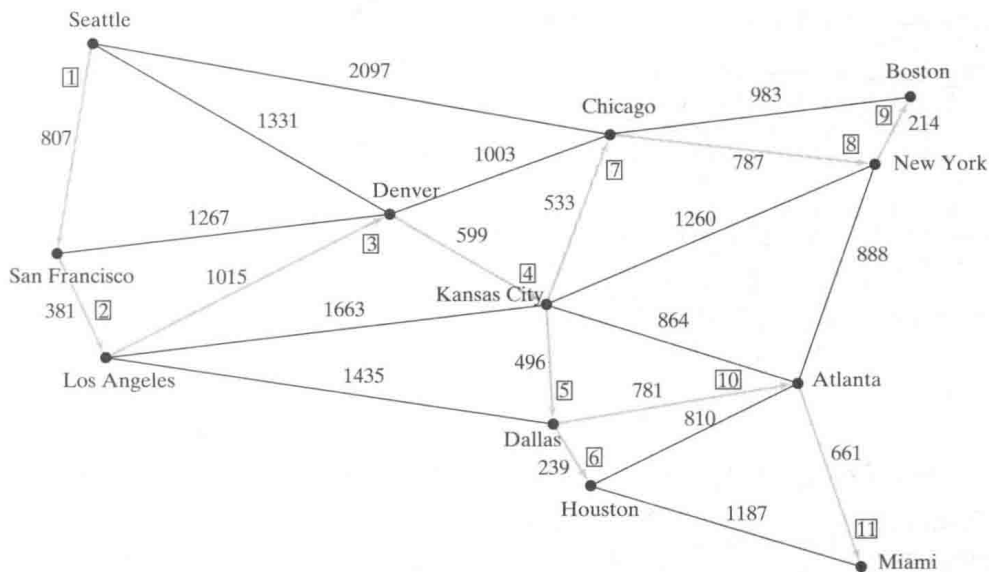
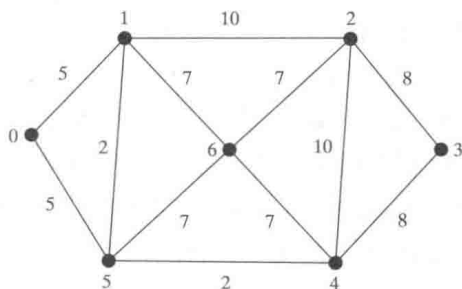


图 29-9 代表城市的最小生成树中的边在图中高亮显示

复习题

29.5 找出下图的一棵最小生成树。



- 29.6 如果所有边的权重不同,那么最小生成树是唯一的吗?
- 29.7 如果使用邻接矩阵来表示加权边,Prim 算法的时间复杂度为多少?
- 29.8 如果图不是连通的,那么 `WeightedGraph` 中的方法 `getMinimumSpanningTree()` 将会怎样? 通过编写一个测试程序,创建一个非连通图并且调用方法 `getMinimumSpanningTree()` 来验证你的答案。如何通过获取局部 MST 来解决这个问题?

29.5 寻找最短路径

🔑 要点提示: 两个顶点之间的最短路径,是指具有最小总权重的路径。

给出一个边的权值非负的图,著名的找出一个单源的最短路径的算法是由荷兰计算机科学家 Edsger Dijkstra 发现的。为了找到从顶点 s 到顶点 v 的最短路径,Dijkstra 的算法寻找从 s 到所有顶点的最短路径。因此 Dijkstra 的算法被称为单源最短路径算法。算法使用 $\text{cost}[v]$ 来存储从顶点 v 到源顶点 s 的最短路径的开销。 $\text{cost}[s]$ 为 0。初始情况下,为所有其他顶点设置 $\text{cost}[v]$ 为无穷大。这个算法重复找出 $V-T$ 中的一个具有最小 $\text{cost}[u]$ 的顶点 u ,并将 u 移到 T 中。

这个算法在程序清单 29-7 中描述。

程序清单 29-7 Dijkstra 的单源最短路径算法

Input: a graph $G = (V, E)$ with non-negative weights

Output: a shortest path tree with the source vertex s as the root

```
1 ShortestPathTree getShortestPath(s) {
2   Let T be a set that contains the vertices whose
3   paths to s are known; Initially T is empty;
4   Set  $\text{cost}[s] = 0$ ; and  $\text{cost}[v] = \text{infinity}$  for all other vertices in V;
5
6   while (size of T < n) {
7     Find u not in T with the smallest  $\text{cost}[u]$ ;
8     Add u to T;
9     for (each v not in T and  $(u, v)$  in E)
10      if ( $\text{cost}[v] > \text{cost}[u] + w(u, v)$ ) {
11         $\text{cost}[v] = \text{cost}[u] + w(u, v)$ ;  $\text{parent}[v] = u$ ;
12      }
13   }
14 }
```

该算法与 Prim 的寻找最小生成树算法非常相似,它们都将顶点分为两个集合 T 和 $V-T$ 。在 Prim 的算法中,集合 T 包含已经添加到树中的顶点。在 Dijkstra 的算法中,集合 T 包含那些已经找到与源顶点之间的最短距离的顶点。这两种算法都重复地从 $V-T$ 中寻找一个顶点,然后将其添加到 T 中。在 Prim 的算法中,这个顶点邻接到集合中某个顶点并具有权重最小边。在 Dijkstra 的算法中,该顶点邻接到集合中某个顶点并具有到源顶点的最小总开销。

算法开始将 $\text{cost}[s]$ 设置为 0 (第 4 行),并为所有其他顶点设置 $\text{cost}[v]$ 为无穷大。然后不断地将顶点(称为 u)从 $V-T$ 添加到 T 中,该顶点具有最小的 $\text{cost}[u]$ (第 7~8 行),如图 29-10a 所示。在顶点 u 被添加到 T 中后,对于每个不在 T 中的 v 顶点,如果 (u, v) 在 T 中并且 $\text{cost}[v] > \text{cost}[u] + w(u, v)$,算法更新 $\text{cost}[v]$ 和 $\text{parent}[v]$ (第 10~11 行)。

使用图 29-11a 中的图来解释 Dijkstra 算法。假设源顶点为顶点 1。因此, $\text{cost}[1]=0$,其他顶点的初始开销为无穷大,如图 29-11b 所示。我们使用 $\text{parent}[i]$ 来表示路径中顶点 i 的父顶点。为了方便起见,设置源结点的父亲为 -1。

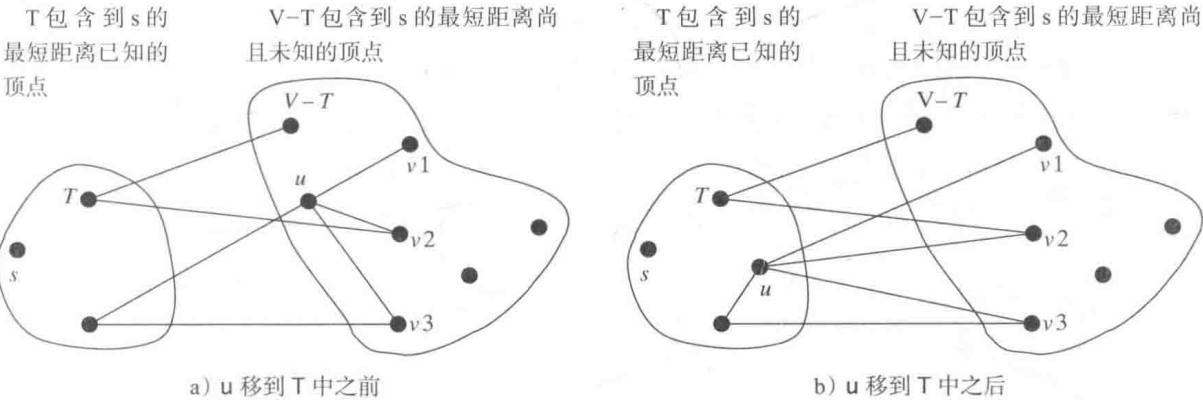


图 29-10 a) 在 V-T 中找到一个具有最小 $\text{cost}[u]$ 的顶点 u ; b) 为每个在 V-T 中并且和 u 相邻的顶点 v 更新 $\text{cost}[v]$

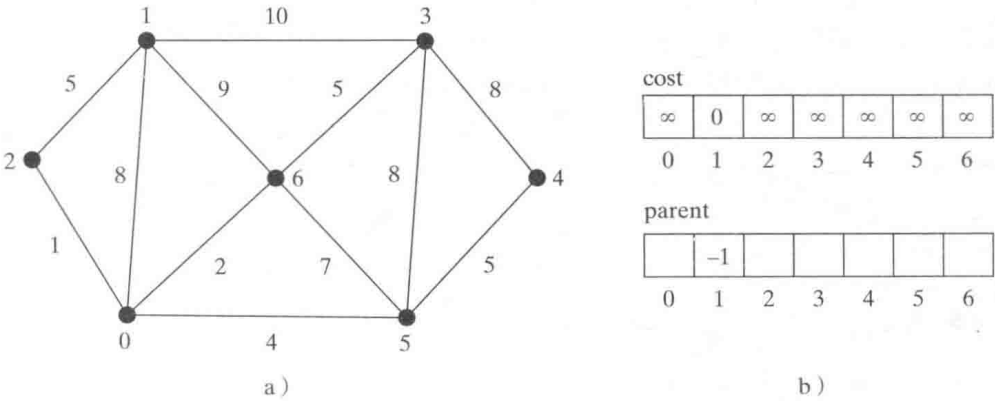


图 29-11 算法将找到从源顶点 1 开始的所有最短路径

初始情况下，集合 T 为空。算法选择具有最小开销的顶点。这种情况下，顶点为 1。算法将 1 添加到 T 中，如图 29-12a 所示。之后，算法为每个和 1 相邻的顶点调整开销值。顶点 2、0、6 和 3 的开销，以及它们的父顶点现在被更新，如图 29-12b 所示。

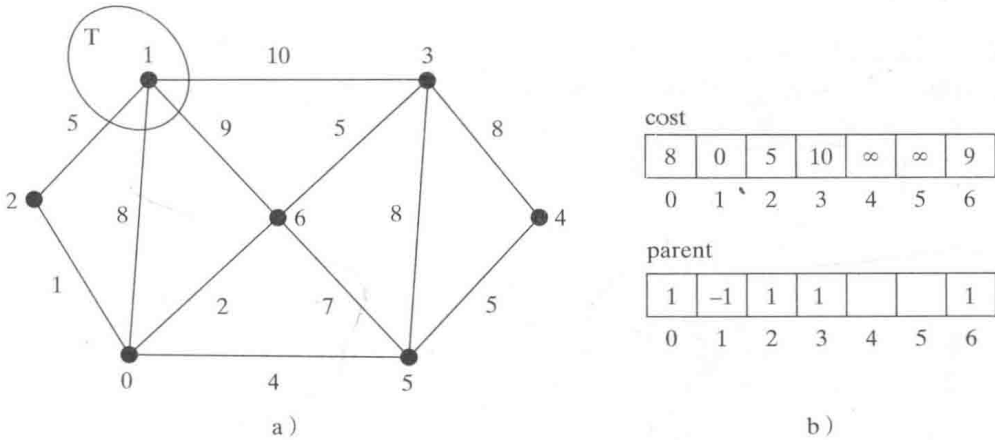


图 29-12 现在顶点 1 在集合 T 中

顶点 2、0、6 和 3 与源顶点相邻，而顶点 2 具有到源顶点 1 的开销最小的路径，于是将顶点 2 添加到 T 中，如图 29-13 所示。更新 $V-T$ 中与 2 相邻的顶点的开销和父顶点。 $\text{cost}[0]$ 现在更新为 6，并且它的父顶点设为 2。从 1 到 2 的箭头表明 2 添加到 T 中之后，1 是 2 的父顶点。

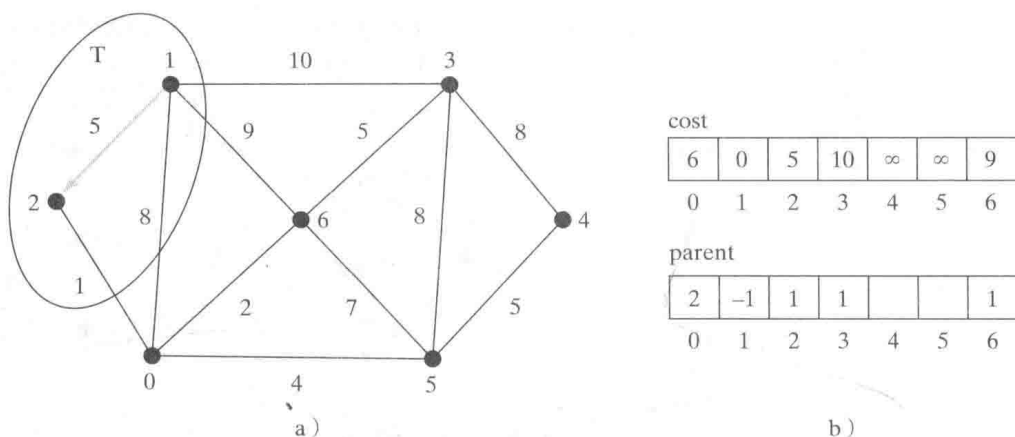
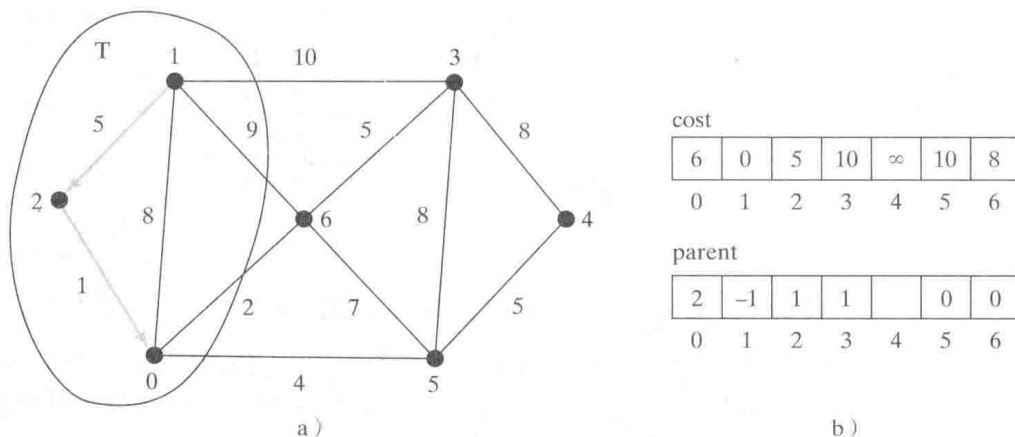
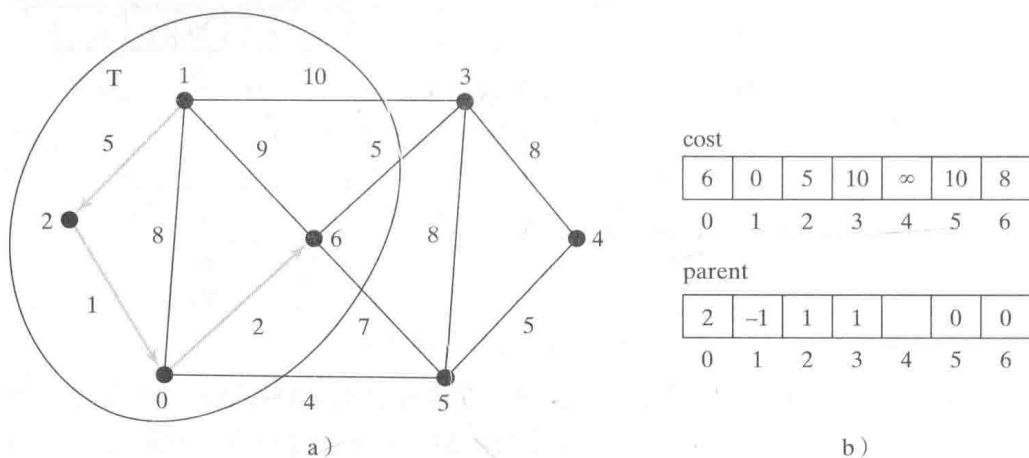


图 29-13 现在顶点 1 和 2 在集合 T 中

现在, T 包含 $\{1, 2\}$ 。在 $V-T$ 中, 顶点 0 具有到源顶点 1 的开销最小的路径, 于是将顶点 0 添加到 T 中, 如图 29-14 所示。更新 $V-T$ 中与 0 相邻的顶点的开销和父顶点。 $\text{cost}[5]$ 现在更新为 10, 并且它的父顶点设为 0; $\text{cost}[6]$ 现在更新为 8, 并且它的父顶点设为 0。

图 29-14 现在顶点 $\{1, 2, 0\}$ 在集合 T 中

现在, T 包含 $\{1, 2, 0\}$ 。在 $V-T$ 中, 顶点 6 具有到源顶点 1 的开销最小的路径, 于是将顶点 6 添加到 T 中, 如图 29-15 所示。更新 $V-T$ 中与 6 相邻的顶点的开销和父顶点。

图 29-15 现在顶点 $\{1, 2, 0, 6\}$ 在集合 T 中

现在， T 包含 $\{1,2,0,6\}$ 。在 $V-T$ 中，顶点 3 和 5 具有最小开销。既可以选择顶点 3，也可以选择顶点 5 放到 T 中。我们将顶点 3 添加到 T 中，如图 29-16 所示。更新 $V-T$ 中与 3 相邻的顶点的开销和父顶点。 $cost[4]$ 现在更新为 18，并且它的父顶点设为 3。

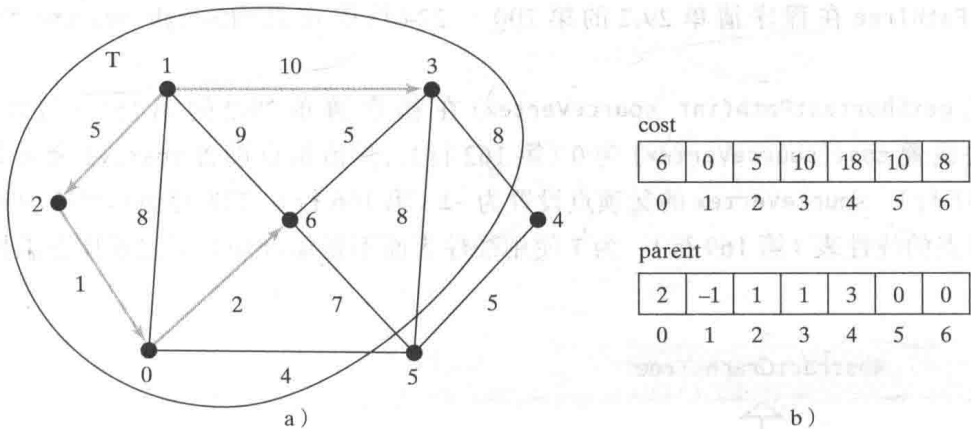


图 29-16 现在顶点 $\{1,2,0,6,3\}$ 在集合 T 中

现在， T 包含 $\{1,2,0,6,3\}$ 。在 $V-T$ 中，顶点 5 具有最小开销，将顶点 5 添加到 T 中，如图 29-17 所示。更新 $V-T$ 中与 5 相邻的顶点的开销和父顶点。 $cost[4]$ 现在更新为 10，并且它的父顶点设为 5。

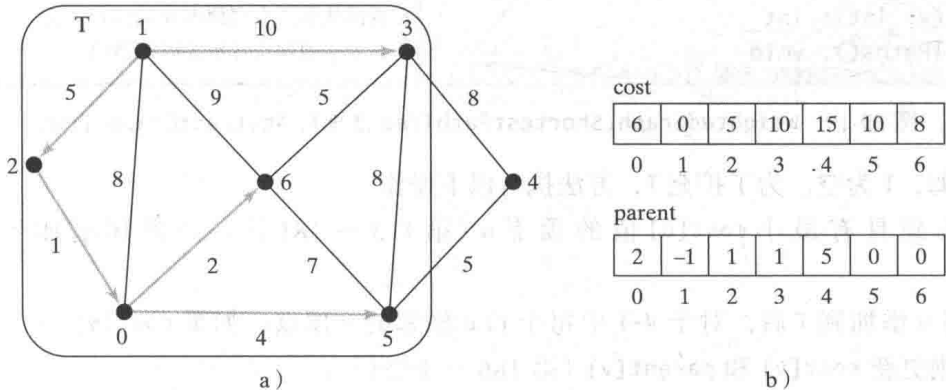


图 29-17 现在顶点 $\{1,2,0,6,3,5\}$ 在集合 T 中

现在， T 包含 $\{1,2,0,6,3,5\}$ 。在 $V-T$ 中，顶点 4 具有最小开销，因此将顶点 4 添加到 T 中，如图 29-18 所示。

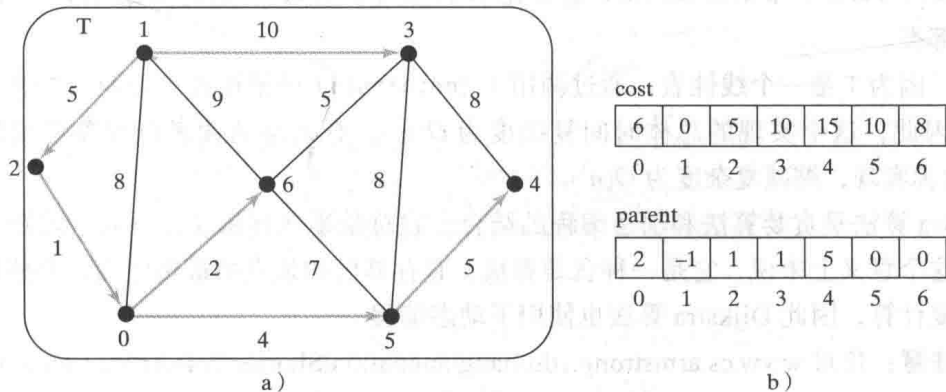


图 29-18 现在顶点 $\{1,2,0,6,3,5,4\}$ 在集合 T 中

正如你所看到的，该算法本质上就是找出从源顶点出发的所有最短路径，它将产生一个以源顶点为根结点的树。我们称这棵树为一棵单源所有最短路径树（或简称最短路径树）。为了建模这棵树，定义一个名为 `ShortestPathTree` 的类继承自 `Tree` 类，如图 29-19 所示。`ShortestPathTree` 在程序清单 29-2 的第 200 ~ 224 行中定义为 `WeightedGraph` 的一个内部类。

方法 `getShortestPath(int sourceVertex)` 在程序清单 29-2 的第 156 ~ 197 行中实现。方法设置 `cost[sourceVertex]` 为 0（第 162 行），其他顶点设置 `cost[v]` 为无穷大（第 159 ~ 161 行）。`sourceVertex` 的父顶点设置为 -1（第 166 行）。`T` 为存储那些添加到最短路径树的顶点的线性表（第 169 行）。为 `T` 使用线性表而不是集合是为了记录顶点添加到 `T` 中的次序。

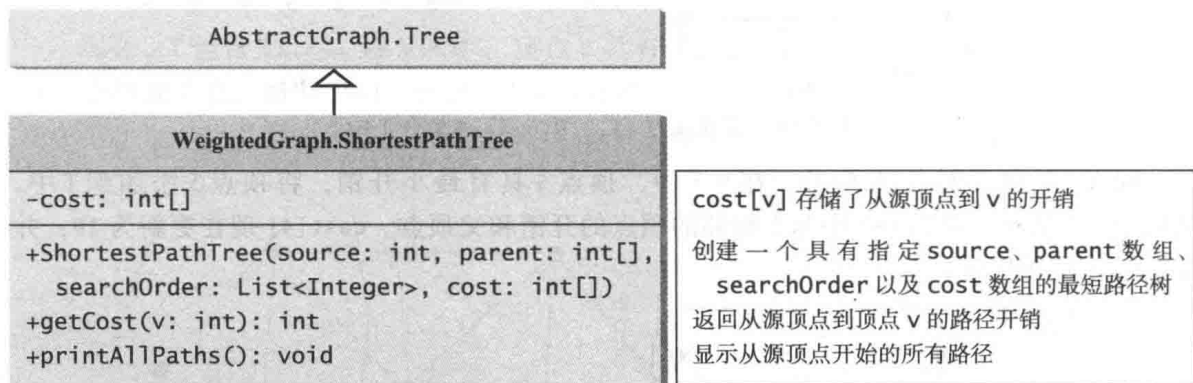


图 29-19 `WeightedGraph.ShortestPathTree` 继承自 `AbstractGraph.Tree`

初始的，`T` 为空。为了扩充 `T`，方法执行以下操作：

- 1) 找到具有最小 `cost[u]` 值的顶点 `u`（第 175 ~ 181 行）并将其添加到 `T` 中（第 183 行）。
- 2) 将 `u` 添加到 `T` 后，对于 `V-T` 中每个和 `u` 相邻的 `v` 顶点，如果 `cost[v] > cost[u] + w(u,v)`，则更新 `cost[v]` 和 `parent[v]`（第 186 ~ 192 行）。

一旦 `s` 的所有顶点都添加到 `T` 中，就会创建一个 `ShortestPathTree` 的实例（第 196 行）。

`ShortestPathTree` 类继承自 `Tree` 类（第 200 行）。为了创建一个 `ShortestPathTree` 的实例，传递 `sourceVertex`、`parent`、`T` 和 `cost`（第 204 ~ 205 行）。`sourceVertex` 成为树的根结点，数据域 `root`、`parent` 和 `searchOrder` 定义在 `Tree` 类中，`Tree` 类是定义在 `AbstractGraph` 中的一个内部类。

注意，因为 `T` 是一个线性表，通过调用 `T.contains(i)` 检测顶点 `i` 是否在 `T` 中需要 $O(n)$ 的时间。因此，这个实现的总体时间复杂度为 $O(n^3)$ 。有兴趣的读者可以参考编程练习题 29.20 来改善实现，缩减复杂度为 $O(n^2)$ 。

Dijkstra 算法是贪婪算法和动态编程的结合。它总是添加到源顶点具有最短距离的新的顶点，从这个意义上来说，它是一种贪婪算法。它存储已知顶点的最短距离，并使用它避免之后的重复计算，因此 Dijkstra 算法也使用了动态编程。

教学注意：使用 www.cs.armstrong.edu/liang/animation/ShortestPathAnimation.html 提供的 GUI 交互式编程，寻找两个城市之间的最短路径，如图 29-20 所示。

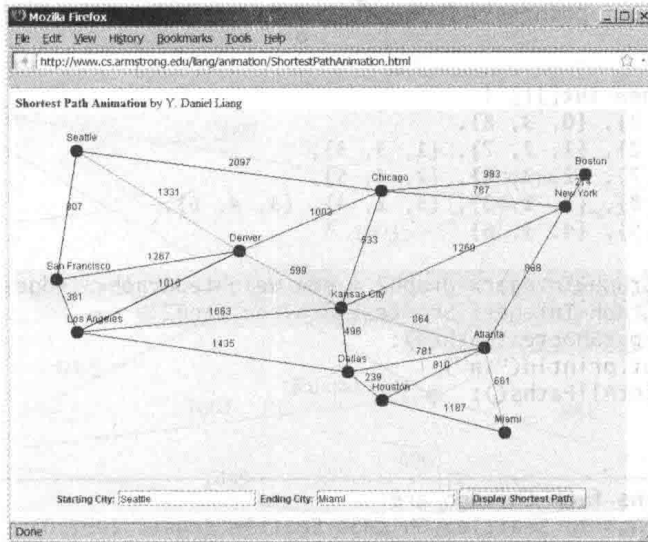


图 29-20 动画工具显示两个城市之间的最短路径

程序清单 29-8 给出了一个测试程序，分别显示图 29-1 中从芝加哥出发到所有其他城市的最短路径，以及图 29-3a 中从顶点 3 到所有顶点的最短路径。

程序清单 29-8 TestShortestPath.java

```

1 public class TestShortestPath {
2     public static void main(String[] args) {
3         String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
4             "Denver", "Kansas City", "Chicago", "Boston", "New York",
5             "Atlanta", "Miami", "Dallas", "Houston"};
6
7         int[][] edges = {
8             {0, 1, 807}, {0, 3, 1331}, {0, 5, 2097},
9             {1, 0, 807}, {1, 2, 381}, {1, 3, 1267},
10            {2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
11            {3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599},
12            {3, 5, 1003},
13            {4, 2, 1663}, {4, 3, 599}, {4, 5, 533}, {4, 7, 1260},
14            {4, 8, 864}, {4, 10, 496},
15            {5, 0, 2097}, {5, 3, 1003}, {5, 4, 533},
16            {5, 6, 983}, {5, 7, 787},
17            {6, 5, 983}, {6, 7, 214},
18            {7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},
19            {8, 4, 864}, {8, 7, 888}, {8, 9, 661},
20            {8, 10, 781}, {8, 11, 810},
21            {9, 8, 661}, {9, 11, 1187},
22            {10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},
23            {11, 8, 810}, {11, 9, 1187}, {11, 10, 239}
24        };
25
26        WeightedGraph<String> graph1 =
27            new WeightedGraph<>(vertices, edges);
28        WeightedGraph<String>.ShortestPathTree tree1 =
29            graph1.getShortestPath(graph1.getIndex("Chicago"));
30        tree1.printAllPaths();
31
32        // Display shortest paths from Houston to Chicago
33        System.out.print("Shortest path from Houston to Chicago: ");
34        java.util.List<String> path
35            = tree1.getPath(graph1.getIndex("Houston"));
36        for (String s: path) {

```

```

37     System.out.print(s + " ");
38 }
39
40 edges = new int[][] {
41     {0, 1, 2}, {0, 3, 8},
42     {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
43     {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
44     {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
45     {4, 2, 5}, {4, 3, 6}
46 };
47 WeightedGraph<Integer> graph2 = new WeightedGraph<>(edges, 5);
48 WeightedGraph<Integer>.ShortestPathTree tree2 =
49     graph2.getShortestPath(3);
50 System.out.println("\n");
51 tree2.printAllPaths();
52 }
53 }

```

```

All shortest paths from Chicago are:
A path from Chicago to Seattle: Chicago Seattle (cost: 2097.0)
A path from Chicago to San Francisco:
    Chicago Denver San Francisco (cost: 2270.0)
A path from Chicago to Los Angeles:
    Chicago Denver Los Angeles (cost: 2018.0)
A path from Chicago to Denver: Chicago Denver (cost: 1003.0)
A path from Chicago to Kansas City: Chicago Kansas City (cost: 533.0)
A path from Chicago to Chicago: Chicago (cost: 0.0)
A path from Chicago to Boston: Chicago Boston (cost: 983.0)
A path from Chicago to New York: Chicago New York (cost: 787.0)
A path from Chicago to Atlanta:
    Chicago Kansas City Atlanta (cost: 1397.0)
A path from Chicago to Miami:
    Chicago Kansas City Atlanta Miami (cost: 2058.0)
A path from Chicago to Dallas: Chicago Kansas City Dallas (cost: 1029.0)
A path from Chicago to Houston:
    Chicago Kansas City Dallas Houston (cost: 1268.0)
Shortest path from Houston to Chicago:
    Houston Dallas Kansas City Chicago

All shortest paths from 3 are:
A path from 3 to 0: 3 1 0 (cost: 5.0)
A path from 3 to 1: 3 1 (cost: 3.0)
A path from 3 to 2: 3 2 (cost: 4.0)
A path from 3 to 3: 3 (cost: 0.0)
A path from 3 to 4: 3 4 (cost: 6.0)

```

程序在第 27 行为图 29-1 创建了一个加权图。然后，调用方法 `getShortestPath` (`graph1.getIndex("Chicago")`) 来返回一个 `Path` 对象，该对象包含从芝加哥出发的所有最短路径。调用 `ShortestPathTree` 对象上的 `printAllPaths()` 显示所有的路径（第 30 行）。

✓ 复习题

- 29.9 追踪 Dijkstra 算法，找到图 29-1 中从波士顿到所有其他城市的最短路径。
- 29.10 如果所有的边都有不同的权重，那么两个顶点之间有最短路径吗？
- 29.11 如果使用邻接矩阵来表示加权边，Dijkstra 算法的时间复杂度是多少？
- 29.12 如果源顶点不能到达图中的所有顶点，那么运行 `WeightedGraph` 中的方法 `getShortestPath()` 将会怎样？编写一个测试程序，创建一个非连通图并且调用方法 `getShortestPath()` 来验证你的答案。如何通过获得一个局部的最短路径树来修改这个问题？
- 29.13 如果没有从顶点 v 到源顶点的路径，`cost[v]` 将是什么？
- 29.14 假设图是连通的；如果 `WeightedGraph` 中的第 159 ~ 161 行删除掉，`getShortestPath` 可以

正确找到最短路径吗？

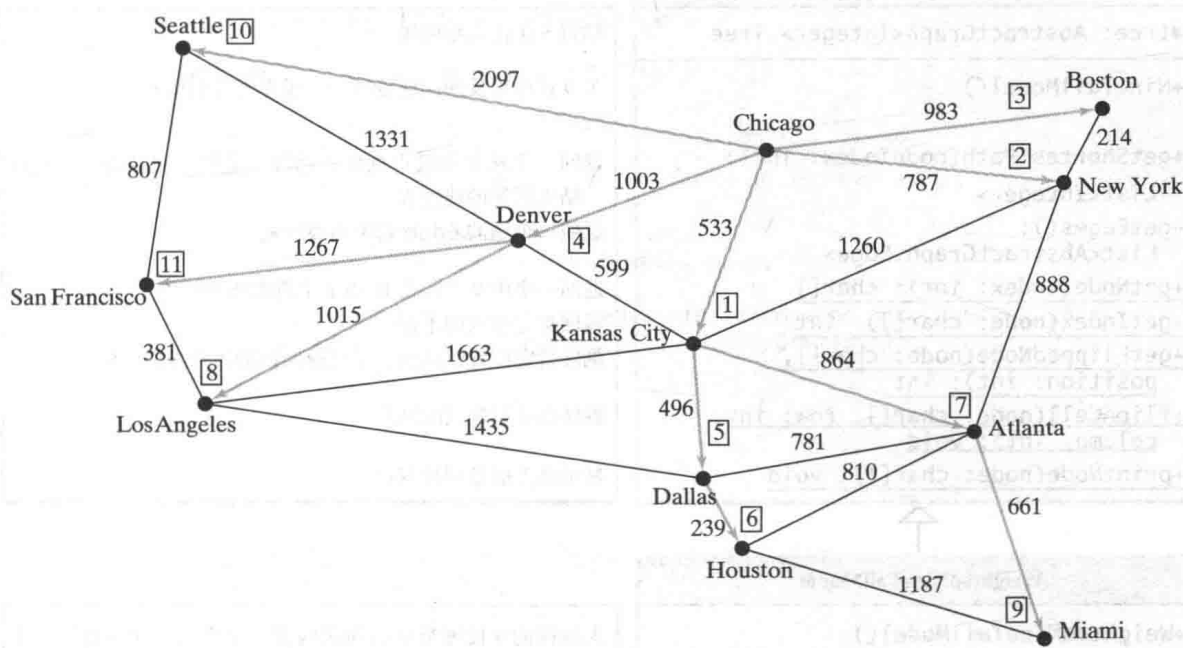


图 29-21 高亮显示从芝加哥到所有其他城市的最短路径

29.6 示例学习：加权的 9 枚硬币反面问题

要点提示：加权的 9 枚硬币反面问题可以简化为加权最短路径问题。
28.10 节中给出了 9 枚硬币反面问题，并且使用广度优先搜索算法解决了这个问题。本节中给出这个问题的变体，并使用最短路径算法解决它。

9 枚硬币反面的问题就是找出使所有的硬币正面朝下的最小数目的移动。每次移动时，翻转一个正面朝上的硬币及其邻居。加权的 9 枚硬币反面问题在每次移动上将翻转的次数指定为权值。例如，可以通过翻转第一行的第一枚硬币和它的两个邻居，将图 29-22a 中的硬币移成图 29-22b 中的状态，因此这次移动的权重为 3。可以通过翻转位于中央的硬币和它的 4 个邻居，将图 29-22c 中的硬币移成图 29-22d 中的状态，因此这次移动的权重为 5。

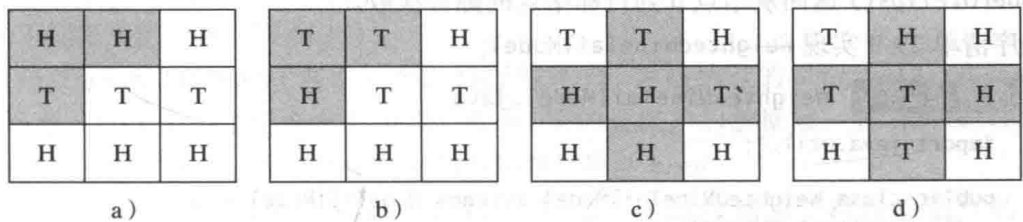


图 29-22 每次移动的权值为该移动的翻转硬币数量

加权的 9 枚硬币反面问题可以简化为在一个边加权图中找出从一个起始结点到目标结点的最短路径。这个图包含 512 个结点。如果存在一个从结点 u 到结点 v 的移动，那么创建一条从结点 v 到结点 u 的边，将翻转的次数指定为边的权重。

回顾一下，在 28.10 节我们定义了一个 `NineTailModel` 类来对 9 枚硬币反面的问题进行建模。现在，我们定义一个名为 `WeightedNineTailModel` 的新类继承自 `NineTailModel`，如图 29-23 所示。

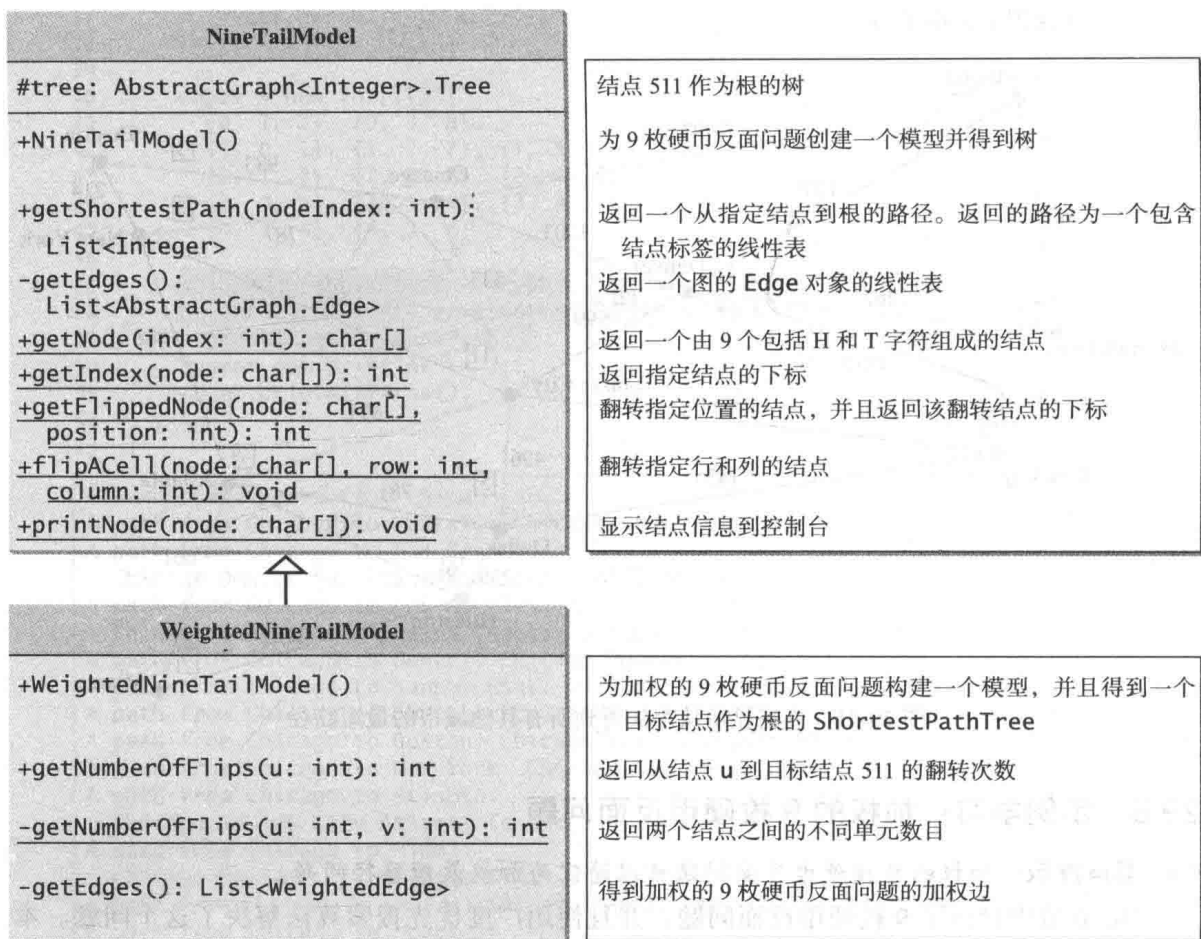


图 29-23 WeightedNineTailModel 继承自 NineTailModel

NineTailModel 类创建一个 Graph 并且获取一个以目标结点 511 为根结点的 Tree。除了创建了一个 WeightedGraph 和获取一个以目标结点 511 为根结点的 ShortestPathTree 外，WeightedNineTailModel 与 NineTailModel 一样。方法 getEdges() 找出图中的所有边。方法 getNumberOfFlips(int u, int v) 返回从结点 u 到结点 v 的翻转次数。方法 getNumberOfFlips() 返回从结点 u 到目标结点的翻转次数。

程序清单 29-9 实现 WeightedNineTailModel。

程序清单 29-9 WeightedNineTailModel.java

```

1  import java.util.*;
2
3  public class WeightedNineTailModel extends NineTailModel {
4      /** Construct a model */
5      public WeightedNineTailModel() {
6          // Create edges
7          List<WeightedEdge> edges = getEdges();
8
9          // Create a graph
10         WeightedGraph<Integer> graph = new WeightedGraph<>
11             (edges, NUMBER_OF_NODES);
12
13         // Obtain a shortest path tree rooted at the target node
14         tree = graph.getShortestPath(511);
15     }

```

```

16
17  /** Create all edges for the graph */
18  private List<WeightedEdge> getEdges() {
19      // Store edges
20      List<WeightedEdge> edges = new ArrayList<>();
21
22      for (int u = 0; u < NUMBER_OF_NODES; u++) {
23          for (int k = 0; k < 9; k++) {
24              char[] node = getNode(u); // Get the node for vertex u
25              if (node[k] == 'H') {
26                  int v = getFlippedNode(node, k);
27                  int numberOfFlips = getNumberOfFlips(u, v);
28
29                  // Add edge (v, u) for a legal move from node u to node v
30                  edges.add(new WeightedEdge(v, u, numberOfFlips));
31              }
32          }
33      }
34
35      return edges;
36  }
37
38  private static int getNumberOfFlips(int u, int v) {
39      char[] node1 = getNode(u);
40      char[] node2 = getNode(v);
41
42      int count = 0; // Count the number of different cells
43      for (int i = 0; i < node1.length; i++)
44          if (node1[i] != node2[i]) count++;
45
46      return count;
47  }
48
49  public int getNumberOfFlips(int u) {
50      return ((WeightedGraph<Integer>.ShortestPathTree)tree)
51          .getCost(u);
52  }
53  }

```

WeightedNineTailModel 继承自 NineTailModel 来创建一个 WeightedGraph, 对加权的 9 枚硬币反面问题进行建模 (第 10 ~ 11 行)。对于每个结点 u , 方法 `getEdges()` 找出一个翻转结点 v , 然后将翻转的次数指定为边 (v, u) 的权重 (第 30 行)。方法 `getNumberOfFlips(int u, int v)` 返回从结点 u 到结点 v 的翻转次数 (第 38 ~ 47 行)。翻转次数是指两个结点之间的不同格子的个数 (第 44 行)。

WeightedNineTailModel 获取一个以目标结点 511 为根结点的 ShortestPathTree (第 14 行)。注意, `tree` 是一个定义在 NineTailModel 中的被保护的数据域, 而 ShortestPathTree 是 Tree 的子类。NineTailModel 中定义的方法使用属性 `tree`。

方法 `getNumberOfFlips(int u)` (第 49 ~ 52 行) 返回从结点 u 到目标结点的翻转次数, 即从结点 u 到目标结点的路径的开销。调用定义在 ShortestPathTree 类中的方法 `getCost(u)` 得到这个开销 (第 51 行)。

程序清单 29-10 给出了一个程序, 提示用户输入一个初始结点并且显示到达目标结点的最小翻转次数。

程序清单 29-10 WeightedNineTail.java

```

1  import java.util.Scanner;
2
3  public class WeightedNineTail {

```



```

4 public static void main(String[] args) {
5     // Prompt the user to enter the nine coins' Hs and Ts
6     System.out.print("Enter an initial nine coins' Hs and Ts: ");
7     Scanner input = new Scanner(System.in);
8     String s = input.nextLine();
9     char[] initialNode = s.toCharArray();
10
11     WeightedNineTailModel model = new WeightedNineTailModel();
12     java.util.List<Integer> path =
13         model.getShortestPath(NineTailModel.getIndex(initialNode));
14
15     System.out.println("The steps to flip the coins are ");
16     for (int i = 0; i < path.size(); i++)
17         NineTailModel.printNode(NineTailModel.getNode(path.get(i)));
18
19     System.out.println("The number of flips is " +
20         model.getNumberOfflips(NineTailModel.getIndex(initialNode)));
21 }
22 }

```

```

Enter an initial nine coins Hs and Ts: HHHTTTTHH Enter

The steps to flip the coins are
HHH
TTT
HHH
HHH
THT
TTT

TTT
TTT
TTT

The number of flips is 8

```

该程序在第 8 行提示用户将一个由 H 和 T 构成的 9 个字母的初始结点作为字符串输入，从该字符串获取一个字符数组（第 9 行），然后创建一个模型（第 11 行），获取从初始结点到目标结点的一个最短路径（第 12 ~ 13 行），显示路径中的结点（第 16 ~ 17 行），最后调用 `getNumberOfflips` 来获取到达目标结点所需的翻转次数（第 20 行）。

✓ 复习题

- 29.15 为什么程序清单 28-13 中 `NineTailModel` 的 `tree` 数据域定义为受保护的？
- 29.16 `WeightedNineTailModel` 中是如何创建图的结点的？
- 29.17 `WeightedNineTailModel` 中是如何创建图的边的？

关键术语

Dijkstra's algorithm (Dijkstra 算法)	shortest path (最短路径)
edge-weighted graph (边加权图)	single-source shortest path (单源最短路径)
minimum spanning tree (最小生成树)	vertex-weighted graph (顶点加权图)
Prim's algorithm (Prim 算法)	

本章小结

1. 可以使用邻接矩阵或者线性表来存储图中的加权边。
2. 图的生成树是一个子图，也是一棵树，并连接着图中所有的顶点。
3. Prim 算法找出最小生成树的工作机制如下：算法首先从包含一个任意结点的生成树开始。算法通过

添加和已在树中的顶点具有最小权重边的结点，来扩展这棵树。

4. Dijkstra 算法从源顶点开始搜索，然后一直寻找到源顶点具有最短路径的结点，直到所有结点被找到。

测试题

回答位于网址 www.cs.armstrong.edu/liang/intro10e/quiz.html 的本章测试题。

编程练习题

- *29.1 (Kruskal 算法) 本书中介绍了找出最小生成树的 Prim 算法。Kruskal 算法是另一种著名的找出最小生成树的算法。该算法重复地找出最小权重边，如果不会形成环，就将它添加到树中。当所有顶点都在树中时，终止这个过程。使用 Kruskal 算法设计和实现一个找出 MST 的算法。
- *29.2 (使用邻接矩阵实现 Prim 算法) 教材在邻接边上使用线性表实现 Prim 算法。对于加权图，使用邻接矩阵实现该算法。
- *29.3 (使用邻接矩阵实现 Dijkstra 算法) 教材在邻接边上使用线性表来实现 Dijkstra 算法。对于加权图，使用邻接矩阵实现该算法。
- *29.4 (修改 9 枚硬币反面问题中的权值) 教材中我们将翻转的次数作为每次移动的权重。假设权值是翻转次数的 3 倍，然后修改这个程序。
- *29.5 (证明或反证) 猜想 `NineTailModel` 和 `WeightedNineTailModel` 可能会得到相同的最短路径。编写程序去证明或者反证这个观点。(提示：令 `tree1` 和 `tree2` 分别表示从 `NineTailModel` 和 `WeightedNineTailModel` 获取的根结点为 511 的树。如果一个结点 `u` 在 `tree1` 中的深度和在 `tree2` 中的深度一样，那么，从结点 `u` 到目标结点的路径长度是相同的。)
- **29.6 (加权 4×4 16 枚硬币反面的模型) 教材中加权的 9 枚硬币反面问题使用的是 3×3 的矩阵。假设你有 16 枚放在 4×4 的矩阵中的硬币。创建一个名为 `WeightedTailModel16` 的新的模型类，然后创建模型的一个实例并且将这个对象存入一个名为 `WeightedTailModel16.dat` 的文件中。
- **29.7 (加权 4×4 16 枚硬币反面问题) 为加权 4×4 16 枚硬币反面的问题修改程序清单 29-9。程序应该读取前一个编程练习题创建的模型对象。
- **29.8 (旅行商人问题) 旅行商人问题 (traveling salesman problem, TSP) 就是找出往返的最短路径，访问每个城市一次且只能访问一次，最后返回到起始城市。这个问题等价于编程练习题 28.17 中的寻找一条最短的哈密尔顿环。在 `WeightedGraph` 类中添加下面的方法：


```
// Return a shortest cycle
// Return null if no such cycle exists
public List<Integer> getShortestHamiltonianCycle()
```
- *29.9 (找出最小生成树) 编写一个程序，从文件中读取一个连通图并且显示它的最小生成树。文件中的第一行是表明顶点个数 (`n`) 的数字。顶点被标记为 `0, 1, ..., n-1`。接下来的每一行以 `u1,v1,w1|u2,v2,w2|...` 的形式来描述边。每个三元组描述一条边和它的权重。图 29-24 显示了与图对应的文件的例子。注意，我们假设图是无向的。如果图有一条边 (u,v) ，那么它也有一条边 (v,u) ，但文件中只表示了一条边。当构建一个图时，两条边都需要考虑。

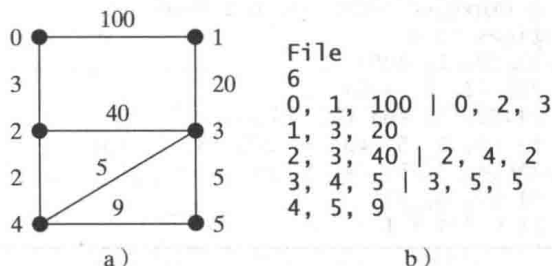


图 29-24 加权图的顶点和边可以存储在一个文件中

程序应该提示用户输入文件名, 然后从文件中读取数据, 建立 `WeightedGraph` 的一个实例 `g`, 调用 `g.printWeightedEdges()` 来显示所有的边, 调用 `getMinimumSpanningTree()` 来获取一个 `WeightedGraph.MST` 的实例 `tree`, 调用 `tree.getTotalWeight()` 来显示最小生成树的权重, 以及调用 `tree.printTree()` 来显示这棵树。下面是这个程序的运行示例:

```
Enter a file name: c:\exercise\WeightedGraphSample.txt Enter
The number of vertices is 6
Vertex 0: (0, 2, 3) (0, 1, 100)
Vertex 1: (1, 3, 20) (1, 0, 100)
Vertex 2: (2, 4, 2) (2, 3, 40) (2, 0, 3)
Vertex 3: (3, 4, 5) (3, 5, 5) (3, 1, 20) (3, 2, 40)
Vertex 4: (4, 2, 2) (4, 3, 5) (4, 5, 9)
Vertex 5: (5, 3, 5) (5, 4, 9)
Total weight in MST is 35
Root is: 0
Edges: (3, 1) (0, 2) (4, 3) (2, 4) (3, 5)
```

提示: 使用 `new WeightedGraph(list, numberOfVertices)` 来创建一个图, 其中 `list` 包含一个 `WeightedEdge` 对象的线性表。使用 `new WeightedEdges(u, v, w)` 来创建一条边。读取第一行以获取顶点的个数。将接下来的每一行读入一个字符串 `s` 中, 并且使用 `s.split("[\\|]")` 来提取三元组。对于每个三元组, `triplet.split(",")` 提取顶点和权值。

***29.10** (为图创建文件) 修改程序清单 29-3, 创建一个表示 `graph1` 的文件。文件格式在编程练习题 29.9 中描述。从程序清单 29-3 中的第 7 ~ 24 行定义的数组来创建文件。图的顶点个数为 12, 它将存储在文件的第一行。如果 $u < v$, 那么存储边 (u, v) 。文件的内容如下所示:

```
12
0, 1, 807 | 0, 3, 1331 | 0, 5, 2097
1, 2, 381 | 1, 3, 1267
2, 3, 1015 | 2, 4, 1663 | 2, 10, 1435
3, 4, 599 | 3, 5, 1003
4, 5, 533 | 4, 7, 1260 | 4, 8, 864 | 4, 10, 496
5, 6, 983 | 5, 7, 787
6, 7, 214
7, 8, 888
8, 9, 661 | 8, 10, 781 | 8, 11, 810
9, 11, 1187
10, 11, 239
```

***29.11** (找出最短路径) 编写一个程序, 从文件中读取一个连通图。图存储在一个文件中, 指定格式与编程练习题 29.9 一样。程序应该提示用户输入文件名、两个顶点, 然后显示这两个顶点之间的最短路径。例如, 对于图 29-23 中的图, 顶点 0 和顶点 1 之间的最短路径可以显示为 0 2 4 3 1。下面是该程序的一个运行示例:

```
Enter a file name: WeightedGraphSample2.txt Enter
Enter two vertices (integer indexes): 0 1 Enter
The number of vertices is 6
Vertex 0: (0, 2, 3) (0, 1, 100)
Vertex 1: (1, 3, 20) (1, 0, 100)
Vertex 2: (2, 4, 2) (2, 3, 40) (2, 0, 3)
Vertex 3: (3, 4, 5) (3, 5, 5) (3, 1, 20) (3, 2, 40)
Vertex 4: (4, 2, 2) (4, 3, 5) (4, 5, 9)
Vertex 5: (5, 3, 5) (5, 4, 9)
A path from 0 to 1: 0 2 4 3 1
```

***29.12** (显示加权图) 修改程序清单 28-6 中的 `GraphView`, 显示加权图。编写一个程序, 显示图 29-1

中的图，如图 29-25 所示。(教师可以要求学生扩展该程序，添加带有正确的边的新的城市到该图中。)

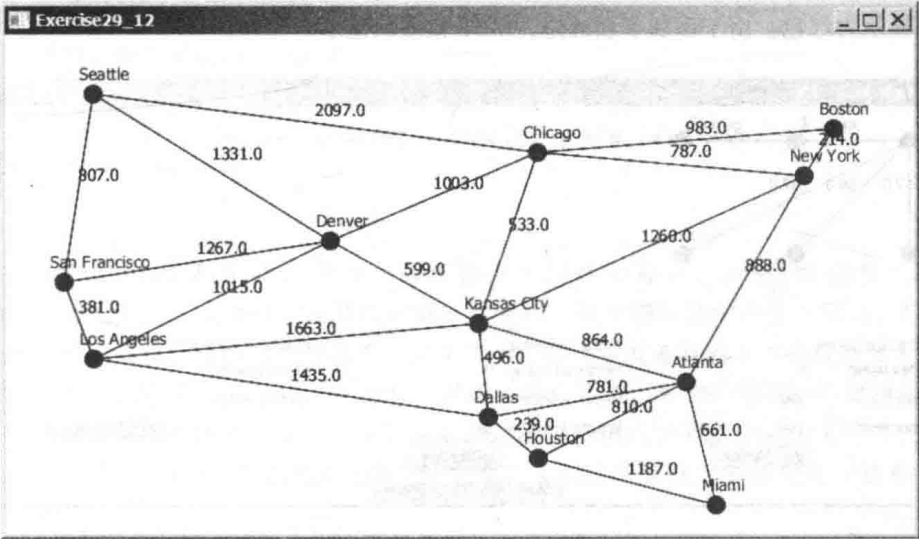


图 29-25 编程练习题 29.12 显示一个加权图

- *29.13 (显示最短路径) 修改程序清单 28-6 中的 GraphView，显示一个加权图和两个指定城市之间的最短路径，如图 29-19 所示。需要在 GraphView 中添加一个数据域 path。如果 path 不为空，路径中的边显示为红色。如果输入了一个图中没有的城市，程序显示一个对话框来警告用户。
- *29.14 (显示最小生成树) 修改程序清单 28-6 中的 GraphView，为图 29-1 中的图显示其加权图和最小生成树，如图 29-26 所示。MST 的路径显示为红色。

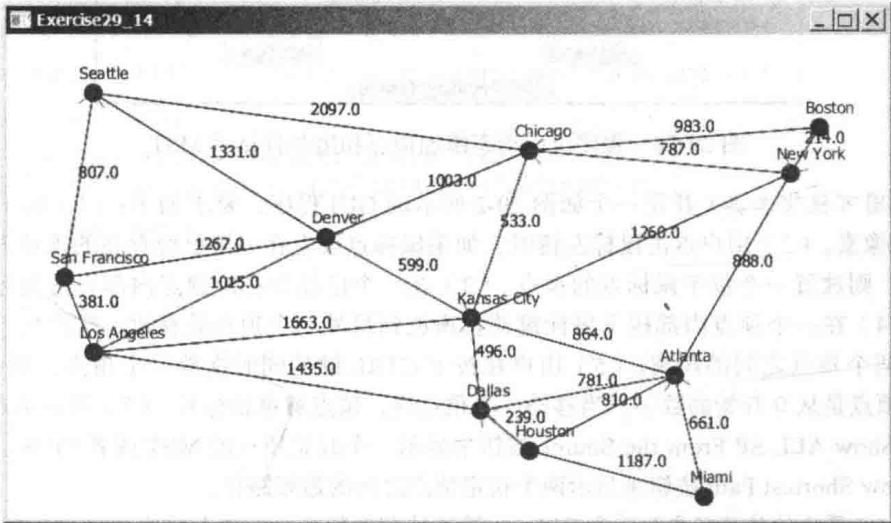


图 29-26 编程练习题 29.14 显示一个 MST

- ***29.15 (动态图) 编写一个程序，允许用户动态创建加权图。用户通过输入顶点的名字和位置来生成顶点，如图 29-27 所示。用户也可以创建一条边来连接两个顶点。为了简化程序，假设顶点的名字和顶点的索引相同。需要以顶点索引顺序 0,1,...,n 来添加顶点。用户可以指定两个顶点并且让程序以红色来显示它们之间的最短路径。

***29.16 (显示一个动态 MST) 编写一个程序, 允许用户动态地创建加权图。用户通过输入顶点的名字和位置来生成顶点, 如图 29-28 所示。用户也可以创建一条边来连接两个顶点。为了简化程序, 假设顶点的名字和顶点的索引相同。需要以顶点索引顺序 $0, 1, \dots, n$ 来添加顶点。MST 的路径显示为红色。由于添加了新的边, MST 被重新显示。

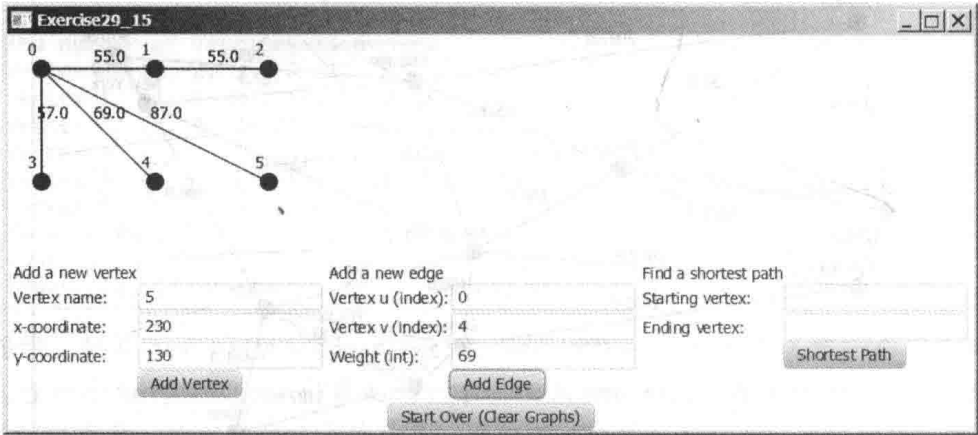


图 29-27 程序可以添加顶点和边并且显示两个指定顶点之间的最短路径

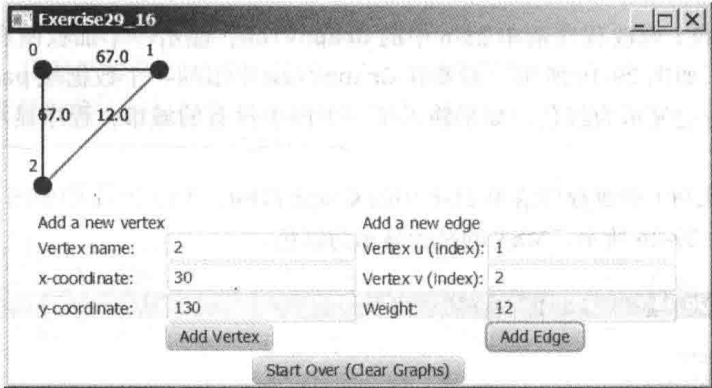


图 29-28 程序可以动态添加顶点和边并且显示 MST

***29.17 (加权图可视化工具) 开发一个如图 29-2 所示的 GUI 程序, 要求如下: (1) 每个顶点的半径为 20 像素。(2) 用户点击鼠标左键时, 如果鼠标点没有在一个已经存在的顶点内部或者过于接近, 则放置一个位于鼠标点的顶点。(3) 在一个已经存在的顶点内部右击鼠标来删除该顶点。(4) 在一个顶点内部按下鼠标键并且拖放到另外一个顶点处释放, 则产生一条边, 并且显示两个顶点之间的距离。(5) 用户在按下 CTRL 键的同时拖放一个顶点, 则移动该顶点。(6) 顶点是从 0 开始的数字。当移动一个顶点时, 顶点被重新标号。(7) 可以单击 Show MST 或者 Show ALL SP From the Source 按钮来显示一个起始顶点的 MST 或者 SP 树。(8) 可以单击 Show Shortest Path 按钮来显示两个指定顶点之间的最短路径。

***29.18 (Dijkstra 算法的替换版本) 一个 Dijkstra 算法的替换版本可以如下描述:

输入: 一个加权图 $G = (V, E)$, 其中权值都为正。

输出: 从一个源顶点 s 开始的最短路径树。

Input: a weighted graph $G = (V, E)$ with non-negative weights
Output: A shortest path tree from a source vertex s

```
1 ShortestPathTree getShortestPath(s) {
```

```

2   Let T be a set that contains the vertices whose
3   paths to s are known;
4   Initially T contains source vertex s with cost[s] = 0;
5   for (each u in V - T)
6       cost[u] = infinity;
7
8   while (size of T < n) {
9       Find v in V - T with the smallest cost[u] + w(u, v) value
10      among all u in T;
11      Add v to T and set cost[v] = cost[u] + w(u, v);
12      parent[v] = u;
13  }
14 }

```

算法使用 $\text{cost}[v]$ 来存储从顶点 v 到源顶点 s 的最短路径。 $\text{cost}[s]$ 为 0。开始时设置 $\text{cost}[v]$ 为无穷大，表示没有找到从 v 到 s 的路径。让 V 表示图中的所有顶点， T 表示已经知道开销的顶点集合。开始时，源顶点 s 位于 T 中。算法重复地找到 T 中的顶点 u 和 $V-T$ 中的顶点 v ，使得 $\text{cost}[u] + w(u, v)$ 最小，并将 v 移至 T 中。教材中给出的最短路径算法不断为 $V-T$ 中的顶点更新开销和父顶点。该算法初始时将每个顶点的开销设置为无穷大，然后在顶点被加入到 T 中的时候仅修改该顶点的开销一次。实现这个算法，并使用程序清单 29-7 来测试你的新算法。

***29.19 (高效找到具有最小 $\text{cost}[u]$ 的顶点 u) `getShortestPath` 方法使用线性搜索，找到具有最小 $\text{cost}[u]$ 的顶点 u 。这将使用 $O(|V|)$ 的时间。搜索时间可以使用一个 AVL 树缩减为 $O(\log |V|)$ 。修改该方法，使用一个 AVL 树来存储 $V-T$ 中的顶点。使用程序清单 29-7 来测试你的新实现。

***29.20 (高效测试一个顶点 u 是否在 T 中) 在程序清单 29-2 中的方法 `getMinimumSpanningTree` 和 `getShortestPath` 中，采用线性表实现了 T 。这样通过调用 `T.contains(u)` 来测试一个顶点 u 是否在 T 中需要 $O(n)$ 的时间。通过引入一个名为 `isInT` 的数组来修改这两个方法。当一个顶点 u 被加入到 T 中的时候设置 `isInT[u]` 为 `true`。测试一个顶点 u 是否在 T 中现在可以在 $O(1)$ 的时间内完成。使用下面的代码编写一个测试程序，其中 `graph1` 是从图 29-1 中创建的。

```

WeightedGraph<String> graph1 = new WeightedGraph<>(edges, vertices);
WeightedGraph<String>.MST tree1 = graph1.getMinimumSpanningTree();
System.out.println("Total weight is " + tree1.getTotalWeight());
tree1.printTree();

WeightedGraph<String>.ShortestPathTree tree2 =
    graph1.getShortestPath(graph1.getIndex("Chicago"));
tree2.printAllPaths();

```

多线程和并行程序设计

【1】教学目标

- 对多线程有一个整体了解 (30.2 节)。
- 通过实现 `Runnable` 接口开发任务类 (30.3 节)。
- 使用 `Thread` 类创建线程以运行任务 (30.3 节)。
- 使用 `Thread` 类中的方法控制线程 (30.4 节)。
- 使用线程来控制动画, 并使用 `Platform.runLater` 来运行应用线程中的代码 (30.5 节)。
- 执行线程池中的任务 (30.6 节)。
- 使用同步方法或阻塞同步线程, 避免竞争状态 (30.7 节)。
- 使用锁来同步线程 (30.8 节)。
- 使用锁的条件来方便线程通信 (30.9 ~ 30.10 节)。
- 使用阻塞队列 (`ArrayBlockingQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue`) 来同步对队列的访问 (30.11 节)。
- 使用信号量限制对共享资源的并行任务的数量 (30.12 节)。
- 使用资源排序技术来避免死锁 (30.13 节)。
- 描述线程的生命周期 (30.14 节)。
- 使用 `Collections` 类中的静态方法创建同步的合集 (30.15 节)。
- 使用 `Fork/Join` 框架实现并行编程 (30.16 节)。

30.1 引言

要点提示: 多线程使得程序中的多个任务可以同时执行。

Java 的重要功能之一就是内部支持多线程——在一个程序中允许同时运行多个任务。在许多程序设计语言中, 多线程都是通过调用依赖于系统的过程或函数来实现的。本章将介绍线程的概念以及如何在 Java 中开发多线程程序。

30.2 线程的概念

要点提示: 一个程序可能包含多个可以同时运行的任务。线程是指一个任务从头至尾的执行流程。

线程提供了运行一个任务的机制。对于 Java 而言, 可以在一个程序中并发地启动多个线程。这些线程可以在多处理器系统上同时运行, 如图 30-1a 所示。

如图 30-1b 所示, 在单处理器系统中, 多个线程共享 CPU 时间称为时间分享, 而操作系统负责调度及分配资源给它们。这种安排是可行的, 因为 CPU 的大部分时间都是空闲的。例如, 在等待用户输入数据时, CPU 什么也不做。

多线程可以使程序反应更快、交互性更强、执行效率更高。例如, 一个好的文字处理程序允许在输入文字的同时, 打印或者保存文件。在一些情况下, 即使在单处理器系统上, 多

线程程序的运行速度也比单线程程序更快。Java 对多线程程序的创建和运行，以及锁定资源以避免冲突提供了非常好的支持。

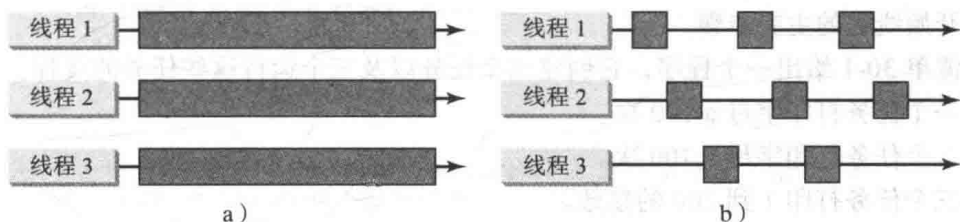


图 30-1 a) 这里多个线程运行在多个 CPU 上；b) 这里多个线程共享单个 CPU

可以在程序中创建附加的线程以执行并发任务。在 Java 中，每个任务都是 `Runnable` 接口的一个实例，也称为可运行对象 (runnable object)。线程本质上讲就是便于任务执行的对象。

复习题

30.1 为什么需要多线程？多线程如何在一个单处理器系统中同时运行？

30.2 什么是可运行对象？线程是什么？

30.3 创建任务和线程

要点提示：一个任务类必须实现 `Runnable` 接口。任务必须从线程运行。

任务就是对象。为了创建任务，必须首先为任务定义一个实现 `Runnable` 接口的类。`Runnable` 接口非常简单，它只包含一个 `run` 方法。需要实现这个方法来说明系统线程将如何运行。开发一个任务类的模板如图 30-2a 所示。

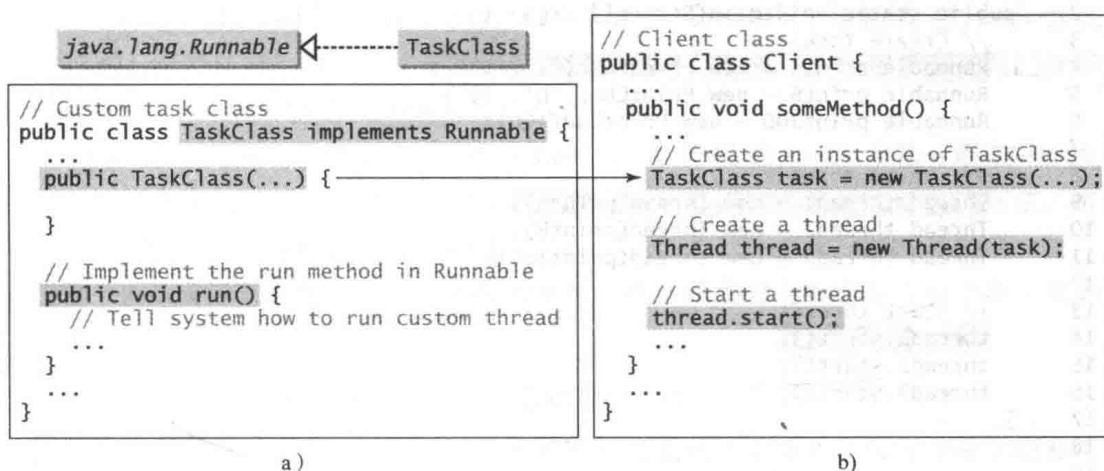


图 30-2 通过实现 `Runnable` 接口定义一个任务类

一旦定义了一个 `TaskClass`，就可以用它的构造方法创建一个任务。例如，

```
TaskClass task = new TaskClass(...);
```

任务必须在线程中执行。`Thread` 类包括创建线程的构造方法以及控制线程的很多有用的方法。使用下面的语句创建任务的线程：

```
Thread thread = new Thread(task);
```

然后调用 `start()` 方法告诉 Java 虚拟机该线程准备运行，如下所示：


```
34     * what task to perform
35     */
36     public void run() {
37         for (int i = 0; i < times; i++) {
38             System.out.print(charToPrint);
39         }
40     }
41 }
42
43 // The task class for printing numbers from 1 to n for a given n
44 class PrintNum implements Runnable {
45     private int lastNum;
46
47     /** Construct a task for printing 1, 2, ..., n */
48     public PrintNum(int n) {
49         lastNum = n;
50     }
51
52     @Override /** Tell the thread how to run */
53     public void run() {
54         for (int i = 1; i <= lastNum; i++) {
55             System.out.print(" " + i);
56         }
57     }
58 }
```

该程序创建了一个任务（第 4～6 行）。为了同时运行它们，创建三个线程（第 9～11 行）。调用 `start()` 方法（第 14～16 行）启动一个线程，它会导致任务中的 `run()` 方法被执行。当 `run()` 方法执行完毕，线程就终止。

因为前两个任务 `printA` 和 `printB` 有类似的功能，所以它们可以定义在同一个任务类 `PrintChar`（第 21～41 行）中。`PrintChar` 类实现 `Runnable`，并且覆盖 `run()` 方法（第 36～40 行），使之具备打印字符动作。该类提供根据给定次数打印任意单个字符的框架。可运行对象 `printA` 和 `printB` 都是 `PrintChar` 类的实例。

`PrintNum` 类（第 44～58 行）实现 `Runnable`，并且覆盖 `run()` 方法（第 53～57 行），使之具备打印数字的动作。该类提供对于任意整数 n ，打印从 1 到 n 的整数的框架。可运行对象 `print100` 是 `PrintNum` 类的一个实例。

注意：如果看不到并发运行三个线程的效果，那么就要增加打印字符的个数。例如，将第 4 行改为

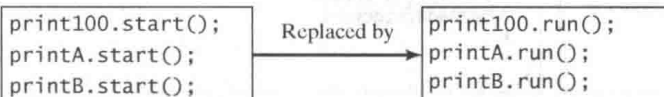
```
Runnable printA = new PrintChar('a', 10000);
```

重要的注意事项：任务中的 `run()` 方法指明如何完成这个任务。Java 虚拟机会自动调用该方法，无需特意调用它。直接调用 `run()` 只是在同一个线程中执行该方法，而没有新线程被启动。

复习题

30.3 如何定义一个任务类？如何为任务创建一个线程？

30.4 如果将程序清单 30-1 中的 14～16 行的 `run()` 方法替换为 `start()` 方法，将会出现什么现象？



30.5 下面两个程序中有什么错误？改正它们。

```
public class Test implements Runnable {
    public static void main(String[] args) {
        new Test();
    }

    public Test() {
        Test task = new Test();
        new Thread(task).start();
    }

    public void run() {
        System.out.println("test");
    }
}
```

```
public class Test implements Runnable {
    public static void main(String[] args) {
        new Test();
    }

    public Test() {
        Thread t = new Thread(this);
        t.start();
        t.start();
    }

    public void run() {
        System.out.println("test");
    }
}
```

30.4 Thread 类

要点提示：Thread 类包含为任务而创建的线程的构造方法，以及控制线程的方法。

图 30-4 给出了 Thread 类的类图。

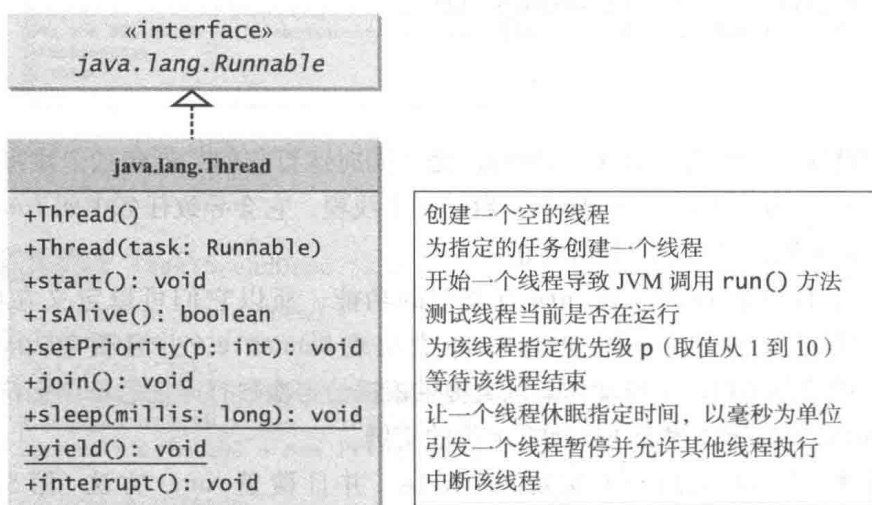


图 30-4 Thread 类包括控制线程的方法

注意：由于 Thread 类实现了 Runnable，所以，可以定义一个 Thread 的扩展类，并且实现 run 方法，如图 30-5a 所示。然后，在客户端程序中创建这个类的一个对象，并且调用它的 start 方法来启动线程，如图 30-5b 所示。

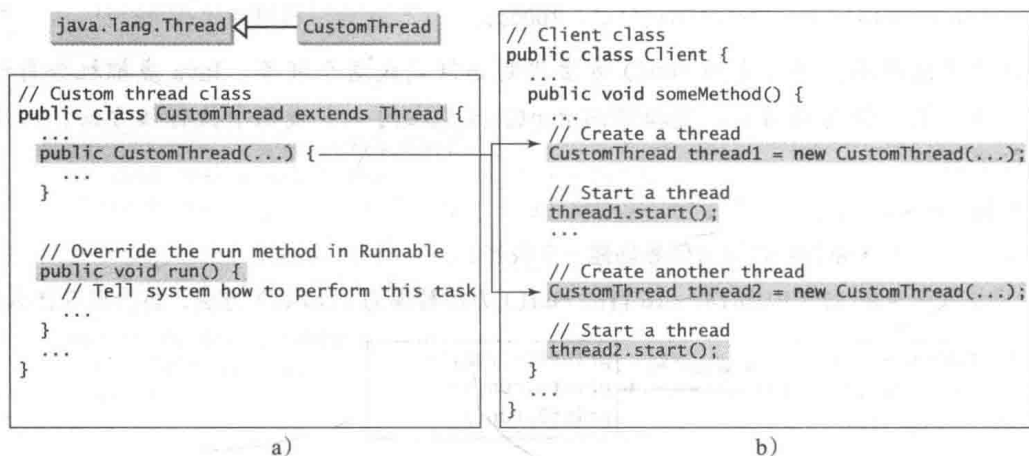


图 30-5 通过继承 Thread 类定义一个线程类

但是，不推荐使用这种方法，因为它将任务和运行任务的机制混在了一起。将任务从线程中分离出来是比较好的设计。

注意：Thread 类还包含方法 `stop()`、`suspend()` 和 `resume()`。由于普遍认为这些方法具有内在的不安全因素，所以，在 Java 2 中不提倡（或不流行）这些方法。为替代方法 `stop()` 的使用，可以通过给 Thread 变量赋值 `null` 来表明它已经停止。

可以使用 `yield()` 方法为其他线程临时让出 CPU 时间。例如，将程序清单 30-1 中 `PrintNum` 类的 `run()` 方法的第 53 ~ 57 行代码做如下修改：

```
public void run() {
    for (int i = 1; i <= lastNum; i++) {
        System.out.print(" " + i);
        Thread.yield();
    }
}
```

每次打印一个数字后，`print100` 任务的线程会让出时间给其他线程。

方法 `sleep(long mills)` 可以将该线程设置为休眠以确保其他线程的执行，休眠时间为指定的毫秒数。例如，程序清单 30-1 中的第 53 ~ 57 行的代码可以修改如下：

```
public void run() {
    try {
        for (int i = 1; i <= lastNum; i++) {
            System.out.print(" " + i);
            if (i >= 50) Thread.sleep(1);
        }
    } catch (InterruptedException ex) {
    }
}
```

每打印一个数字 (≥ 50) 之后，`print100` 任务的线程休眠 1 毫秒。

`sleep` 方法可能抛出一个 `InterruptedException`，这是一个必检异常。当一个休眠线程的 `interrupt()` 方法被调用时，就会发生这样的一个异常。这个 `interrupt()` 方法极少在线程上被调用，所以，不太可能发生 `InterruptedException` 异常。但是，因为 Java 强制捕获必检的异常，所以，必须将它放到 `try-catch` 块中。如果在一个循环中调用了 `sleep` 方法，那就应该将这个循环放在 `try-catch` 块中，如下面图 a 所示。如果循环在 `try-catch` 块外，如图 b 所示，即使线程被中断，它也可能会继续执行。

```
public void run() {
    try {
        while (...) {
            ...
            Thread.sleep(1000);
        }
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
}
```

a) 正确

```
public void run() {
    while (...) {
        try {
            ...
            Thread.sleep(sleepTime);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}
```

b) 不正确

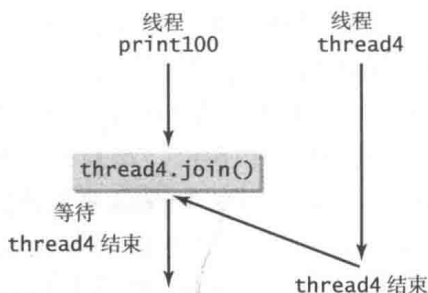
可以使用 `join()` 方法使一个线程等待另一个线程的结束。例如，假设对程序清单 30-1 中的第 53 ~ 57 行代码做如下修改：

创建了一个新线程 `thread4`。它打印字符 `c` 40 次。在线程 `thread4` 结束后打印从 50 到 100 的数字。

```

public void run() {
    Thread thread4 = new Thread(
        new PrintChar('c', 40));
    thread4.start();
    try {
        for (int i = 1; i <= lastNum; i++) {
            System.out.print(" " + i);
            if (i == 50) thread4.join();
        }
    } catch (InterruptedException ex) {}
}

```



Java 给每个线程指定一个优先级。默认情况下，线程继承生成它的线程的优先级。可以用 `setPriority` 方法提高或降低线程的优先级，还能用 `getPriority` 方法获取线程的优先级。优先级是从 1 到 10 的数字。Thread 类有 `int` 型常量 `MIN_PRIORITY`、`NORM_PRIORITY` 和 `MAX_PRIORITY`，分别代表 1、5 和 10。主线程的优先级是 `Thread.NORM_PRIORITY`。

Java 虚拟机总是选择当前优先级最高的可运行线程。较低优先级的线程只有在没有比它更高的优先级的线程运行时才能运行。如果所有可运行线程具有相同的优先级，那将会用循环队列给它们分配相同的 CPU 份额。这被称为循环调度（round-robin scheduling）。例如，假设在程序清单 30-1 的第 16 行插入下面的代码：

```
thread3.setPriority(Thread.MAX_PRIORITY);
```

则任务 `print100` 的线程首先结束。

提示：在 Java 将来的版本中，优先级的数字可能会改变。为将这种变化带来的影响降低到最低，可以使用 Thread 类中的常量来指定线程优先级。

提示：如果总有一个优先级较高的线程在运行，或者有一个相同优先级的线程不退出，那么这个线程可能永远也没有运行的机会。这种情况称为资源竞争或缺乏（contention or starvation）。为避免竞争现象，高优先级的线程必须定时地调用 `sleep` 方法或 `yield` 方法，来给低优先级或相同优先级的线程一个运行的机会。

复习题

30.6 下面哪些方法是 `java.lang.Thread` 中的实例方法？哪些方法可能抛出异常 `InterruptedException`？哪些方法在 Java 中是禁用的？

`run`, `start`, `stop`, `suspend`, `resume`, `sleep`, `interrupt`, `yield`, `join`

30.7 如果循环中包含抛出 `InterruptedException` 异常的方法，那么为什么这个循环必须放在 `try-catch` 块中？

30.8 如何设置线程的优先级？线程的默认优先级是什么？

30.5 示例学习：闪烁的文本

要点提示：可以使用线程来控制动画。

15.11 节中介绍过使用 `Timeline` 对象控制动画。也可以使用线程来控制动画。程序清单 30-2 给出如何在一个标签上显示闪烁文本，如图 30-6 所示。

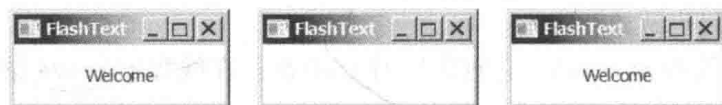


图 30-6 文本 Welcome 闪烁

程序清单 30-2 FlashText.java

```

1  import javafx.application.Application;
2  import javafx.application.Platform;
3  import javafx.scene.Scene;
4  import javafx.scene.control.Label;
5  import javafx.scene.layout.StackPane;
6  import javafx.stage.Stage;
7
8  public class FlashText extends Application {
9      private String text = "";
10
11     @Override // Override the start method in the Application class
12     public void start(Stage primaryStage) {
13         StackPane pane = new StackPane();
14         Label lblText = new Label("Programming is fun");
15         pane.getChildren().add(lblText);
16
17         new Thread(new Runnable() {
18             @Override
19             public void run() {
20                 try {
21                     while (true) {
22                         if (lblText.getText().trim().length() == 0)
23                             text = "Welcome";
24                         else
25                             text = "";
26
27                         Platform.runLater(new Runnable() { // Run from JavaFX GUI
28                             @Override
29                             public void run() {
30                                 lblText.setText(text);
31                             }
32                         });
33
34                         Thread.sleep(200);
35                     }
36                 }
37                 catch (InterruptedException ex) {
38                 }
39             }
40         }).start();
41
42         // Create a scene and place it in the stage
43         Scene scene = new Scene(pane, 200, 50);
44         primaryStage.setTitle("FlashText"); // Set the stage title
45         primaryStage.setScene(scene); // Place the scene in the stage
46         primaryStage.show(); // Display the stage
47     }
48 }

```

程序在一个匿名内部类中创建了一个 `Runnable` 对象（第 17 ~ 40 行）。这个对象在 40 行启动并持续地运行以修改标签中的文本。如果标签为空白的，则设置为文本（第 23 行），如果标签具有一个文本，则设置为空白（第 25 行）。通过设置和取消文本来模拟一个闪烁的效果。

JavaFX GUI 运行自 JavaFX 应用程序线程。闪烁的控制运行自一个单独的线程。非应用程序线程中的代码不能更新应用程序线程中的 GUI。为了更新标签中的文本，第 27 ~ 32 行创建了一个新的 `Runnable` 对象。调用 `Platform.runLater(Runnable r)` 告诉系统在应用程序线程中运行 `Runnable` 对象。

可以使用 `lambda` 表达式简化程序中匿名内部类，如下所示：


```

new Thread(() -> { // lambda expression
    try {
        while (true) {
            if (lblText.getText().trim().length() == 0)
                text = "Welcome";
            else
                text = "";

            Platform.runLater(() -> lblText.setText(text)); // lambda exp
            Thread.sleep(200);
        }
    } catch (InterruptedException ex) {
    }
}).start();

```


✓ 复习题

- 30.9 什么导致了文本的闪烁?
- 30.10 FlashText 的实例是一个可运行对象吗?
- 30.11 使用 Platform.runLater 的目的是什么?
- 30.12 可以将第 27 ~ 32 行的代码替换为如下代码吗?

```
Platform.runLater(e -> lblText.setText(text));
```

- 30.13 如果没有应用第 34 行 (Thread.sleep(200)) 会发生什么?

30.6 线程池

 **要点提示:** 可以使用线程池来高效执行任务。

30.3 节中学习了如何实现 java.lang.Runnable 来定义一个任务类, 以及如何创建一个线程来运行一个任务, 如下所示:

```

Runnable task = new TaskClass(task);
new Thread(task).start();

```

该方法对单一任务的执行是很方便的, 但是由于必须为每个任务创建一个线程, 因此对大量的任务而言是不够高效的。为每个任务开始一个新线程可能会限制吞吐量并且造成性能降低。线程池是管理并发执行任务个数的理想方法。Java 提供 Executor 接口来执行线程池中的任务, 提供 ExecutorService 接口来管理和控制任务。ExecutorService 是 Executor 的子接口, 如图 30-7 所示。

为了创建一个 Executor 对象, 可以使用 Executors 类中的静态方法, 如图 30-8 所示。newFixedThreadPool(int) 方法在池中创建固定数目的线程。如果线程完成了任务的执行, 它可以被重新使用以执行另外一个任务。如果线程池中所有的线程都不是处于空闲状态, 而且有任务在等待执行, 那么在关闭之前, 如果由于一个错误终止了一个线程, 就会创建一个新线程来替代它。如果线程池中所有的线程都不是处于空闲状态, 而且有任务在等待执行, 那么 newCachedThreadPool() 方法就会创建一个新线程。如果缓冲池中的线程在 60 秒内都没有被使用就该终止它。对许多小任务而言, 一个缓冲池已经足够。

程序清单 30-3 显示如何使用线程池改写程序清单 30-1。

第 6 行创建了一个最大线程数为 3 的线程池执行器。在程序清单 30-1 中定义了类 PrintChar 和 PrintNum。第 9 行创建任务 new PrintChar('a', 100), 并且将它添加到线程池中。在第 10 和 11 行, 创建了另外两个可运行的任务, 并且将它们添加到同一个线程池

中。执行器创建三个线程来并发执行三个任务。

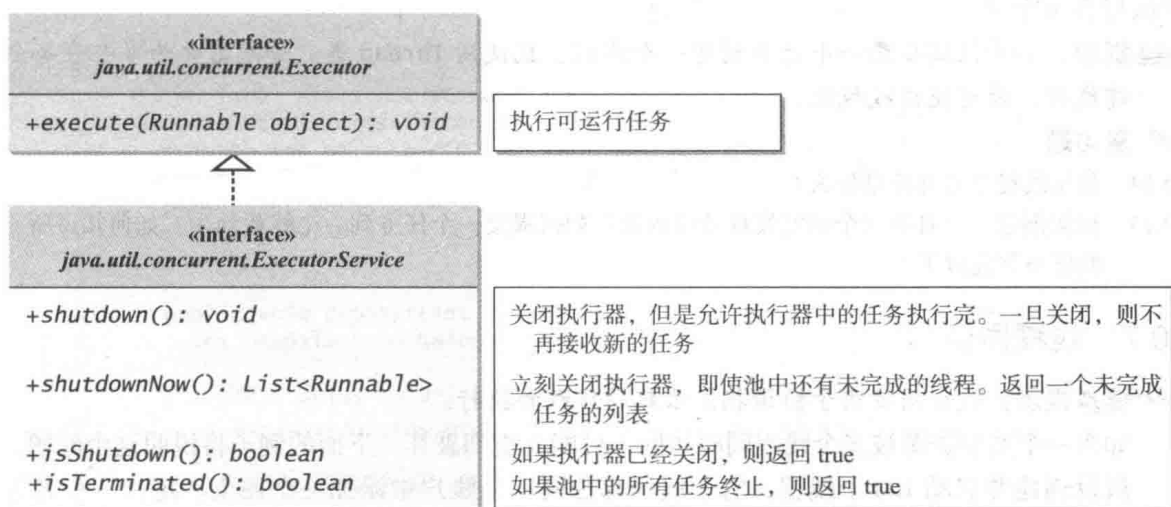


图 30-7 Executor 接口执行线程，而子接口 ExecutorService 管理线程

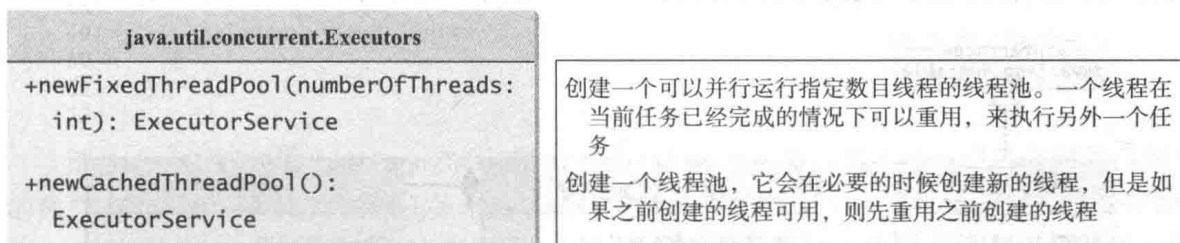


图 30-8 Executors 类提供创建 Executor 对象的静态方法

程序清单 30-3 ExecutorDemo.java

```

1 import java.util.concurrent.*;
2
3 public class ExecutorDemo {
4     public static void main(String[] args) {
5         // Create a fixed thread pool with maximum three threads
6         ExecutorService executor = Executors.newFixedThreadPool(3);
7
8         // Submit runnable tasks to the executor
9         executor.execute(new PrintChar('a', 100));
10        executor.execute(new PrintChar('b', 100));
11        executor.execute(new PrintNum(100));
12
13        // Shut down the executor
14        executor.shutdown();
15    }
16 }
  
```

如果用下面的语句替换第 6 行

```
ExecutorService executor = Executors.newFixedThreadPool(1);
```

会发生什么呢？这三个可运行的任务将顺次执行，因为在线程池中只有一个线程。

如果将第 6 行用下面的语句替换，

```
ExecutorService executor = Executors.newCachedThreadPool();
```

又会发生什么呢？将为每个等待的任务创建一个新线程，所以，所有的任务都并发地执行。

第 14 行的方法 `shutdown()` 通知执行器关闭。不能接受新的任务，但是现有的任务将继续执行直至完成。

[{}] 提示：如果仅需要为一个任务创建一个线程，就使用 `Thread` 类。如果需要为多个任务创建线程，最好使用线程池。

✓ 复习题

30.14 使用线程池的好处是什么？

30.15 如何创建一个具有三个固定线程的线程池？如何提交一个任务到一个线程池中？如何知道所有的任务都完成了？

30.7 线程同步

要点提示：线程同步用于协调相互依赖的线程的执行。

如果一个共享资源被多个线程同时访问，可能会遭到破坏。下面的例子将说明这个问题。

假设创建并启动 100 个线程，每个线程都往同一个账户中添加一个便士。定义一个名为 `Account` 的类模拟账户，一个名为 `AddAPennyTask` 的类用来向账户里添加一便士，以及一个用于创建和启动线程的主类。这些类之间的关系如图 30-9 所示。程序清单 30-4 给出了这个程序。

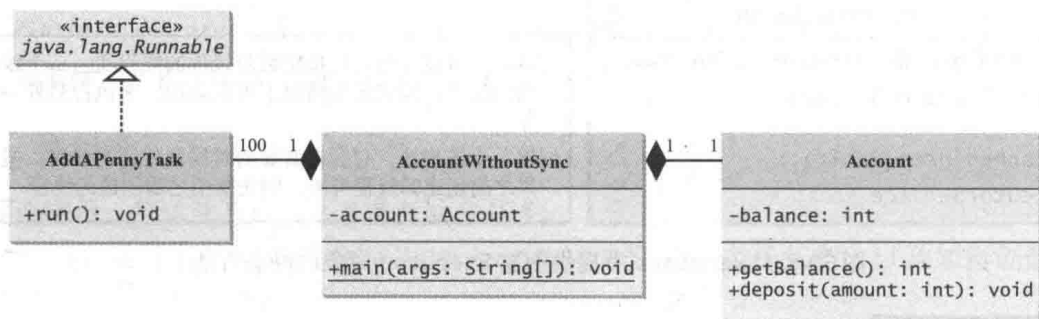


图 30-9 `AccountWithoutSync` 包含一个 `Account` 的实例和 `AddAPennyTask` 的 100 个线程

程序清单 30-4 `AccountWithoutSync.java`

```

1  import java.util.concurrent.*;
2
3  public class AccountWithoutSync {
4      private static Account account = new Account();
5
6      public static void main(String[] args) {
7          ExecutorService executor = Executors.newCachedThreadPool();
8
9          // Create and launch 100 threads
10         for (int i = 0; i < 100; i++) {
11             executor.execute(new AddAPennyTask());
12         }
13
14         executor.shutdown();
15
16         // Wait until all tasks are finished
17         while (!executor.isTerminated()) {
18             ;
19         }
20
21         System.out.println("What is balance? " + account.getBalance());
22     }
23
24     // A thread for adding a penny to the account
25     private static class AddAPennyTask implements Runnable {

```

```
25     public void run() {
26         account.deposit(1);
27     }
28 }
29
30 // An inner class for account
31 private static class Account {
32     private int balance = 0;
33
34     public int getBalance() {
35         return balance;
36     }
37
38     public void deposit(int amount) {
39         int newBalance = balance + amount;
40
41         // This delay is deliberately added to magnify the
42         // data-corruption problem and make it easy to see.
43         try {
44             Thread.sleep(5);
45         }
46         catch (InterruptedException ex) {
47         }
48
49         balance = newBalance;
50     }
51 }
52 }
```

第 24 ~ 51 行的类 `AddAPennyTask` 和 `Account` 都是内部类。第 4 行创建具有初始余额 0 的账户 `Account`。第 11 行创建任务来给该账户增加一个便士，并且将该任务提交给执行器。第 11 行在第 10 ~ 12 行重复 100 次。第 17 和 18 行中程序重复检验所有任务是否完成。第 20 行显示所有任务完成之后的账户余额。

程序创建 100 个在线程池 `executor` 中执行的线程（第 10 ~ 12 行），方法 `isTerminated()`（第 17 行）被用来测试线程是否终止。

该账户中的初始余额为 0（第 32 行）。当所有的线程都完成时，余额应该是 100，但是输出结果并不是可预测的。正如图 30-10 所示，运行样例中的答案是错误的。它演示了当所有线程同时访问同一个数据源时，就会出现数据破坏的问题。

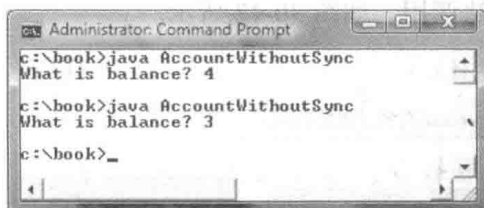


图 30-10 `AccountWithoutSync` 程序导致了数据的不一致性

第 39 ~ 49 行可以用下面的语句代替：

```
balance = balance + amount;
```

使用这条语句不大可能重现刚才出现的问题。设计第 39 ~ 49 行的语句是为了故意放大数据破坏的可能性，使它更容易显现出来。如果运行了几遍程序还没有看出问题，可以增加第 44 行的休眠时间。这会显著地增加出现数据不一致问题的可能性。

那么，究竟是什么导致了程序的错误？下面给出一个可能的情景，如图 30-11 所示。

Step	Balance	Task 1	Task 2
1	0	<code>newBalance = balance + 1;</code>	
2	0		<code>newBalance = balance + 1;</code>
3	1	<code>balance = newBalance;</code>	
4	1		<code>balance = newBalance;</code>

图 30-11 任务 1 和任务 2 都向同一余额里加 1

在步骤 1 中，任务 1 从账户中获取余额数目。在步骤 2 中，任务 2 从账户中获取同样数目的余额。在步骤 3 中，任务 1 向账户写入一个新余额。在步骤 4 中，任务 2 也向该账户写入一个新余额。

这个情景的效果就是任务 1 什么也没做，因为在步骤 4 中，任务 2 覆盖了任务 1 的结果。很明显，问题是任务 1 和任务 2 以一种会引起冲突的方式访问一个公共资源。这是多线程程序中的一个普遍问题，称为竞争状态（race condition）。如果一个类的对象在多线程程序中没有导致竞争状态，则称这样的类为线程安全的（thread-safe）。如上例所示，Account 类不是线程安全的。

30.7.1 synchronized 关键字

为避免竞争状态，应该防止多个线程同时进入程序的某一特定部分，程序中的这部分称为临界区（critical region）。程序清单 30-4 中的临界区是整个 deposit 方法。可以使用关键字 synchronized 来同步方法，以便一次只有一个线程可以访问这个方法。有几种办法可以解决程序清单 30-4 中的问题。一种办法是通过在第 38 行的 deposit 方法中添加关键字 synchronized，使 Account 类成为线程安全的，如下所示：

```
public synchronized void deposit(double amount)
```

一个同步方法在执行之前需要加锁。锁是一种实现资源排他使用的机制。对于实例方法，要给调用该方法的对象加锁。对于静态方法，要给这个类加锁。如果一个线程调用一个对象上的同步实例方法（静态方法），首先给该对象（类）加锁，然后执行该方法，最后解锁。在解锁之前，另一个调用那个对象（类）中方法的线程将被阻塞，直到解锁。

将 deposit 方法同步后，前面的情景不会再出现。如果任务 1 进入了方法，任务 2 将被阻塞，直到任务 1 结束了方法调用，如图 30-12 所示。

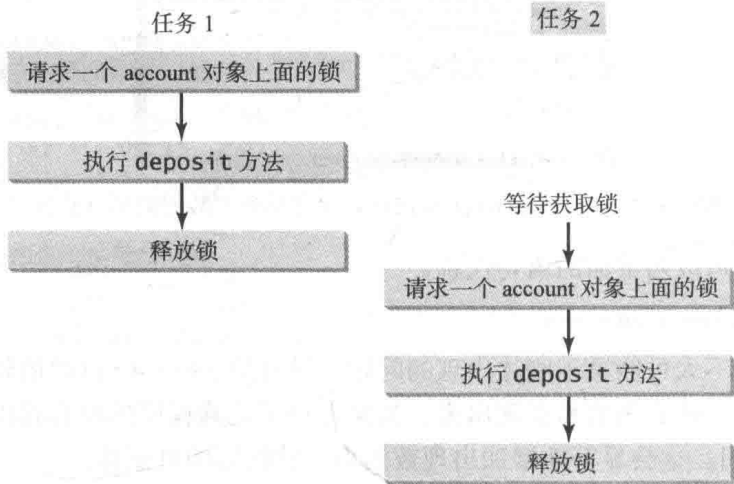


图 30-12 任务 1 和任务 2 被同步

30.7.2 同步语句

调用一个对象上的同步实例方法，需要给该对象加锁。而调用一个类上的同步静态方法，需要给该类加锁。当执行方法中某一个代码块时，同步语句不仅可用于对 `this` 对象加锁，而且可用于对任何对象加锁。这个代码块称为同步块（`synchronized block`）。同步语句的一般形式如下所示：

```
synchronized (expr) {  
    statements;  
}
```

表达式 `expr` 求值结果必须是一个对象的引用。如果对象已经被另一个线程锁定，则在解锁之前，该线程将被阻塞。当获准对一个对象加锁时，该线程执行同步块中的语句，然后解除给对象所加的锁。

同步语句允许设置同步方法中的部分代码，而不必是整个方法。这大大增强了程序的并发能力。将第 26 行的语句放入同步块中，可以把程序清单 30-4 改成线程安全的：

```
synchronized (account) {  
    account.deposit(1);  
}
```

注意：任何同步的实例方法都可以转换为同步语句。例如，下图 a 中的同步实例方法等价于图 b 中的同步实例方法：

```
public synchronized void xMethod() {  
    // method body  
}
```

```
public void xMethod() {  
    synchronized (this) {  
        // method body  
    }  
}
```

✓ 复习题

30.16 给出一些运行多个线程时会产生资源破坏的示例。如何同步有冲突的线程？

30.17 假设将程序清单 30-4 中第 26 行的语句放到一个同步块中来避免竞争状态，如下所示：

```
synchronized (this) {  
    account.deposit(1);  
}
```

这样可行吗？

30.8 利用加锁同步

要点提示：可以显式地采用锁和状态来同步线程。

回顾一下，在程序清单 30-4 中，100 个任务向同一个账户并发存储一个便士，这会造成冲突。在 `deposit` 方法中使用 `synchronized` 关键字可以避免这种情况，如下所示：

```
public synchronized void deposit(double amount)
```

同步的实例方法在执行方法之前都隐式地需要一个加在实例上的锁。

Java 可以显式地加锁，这给协调线程带来了更多的控制功能。一个锁是一个 `Lock` 接口的实例，它定义了加锁和释放锁的方法，如图 30-13 所示。锁也可以使用 `newCondition()` 方法来创建任意个数的 `Condition` 对象，用来进行线程通信。

`ReentrantLock` 是 `Lock` 的一个具体实现，用于创建相互排斥的锁。可以创建具有特定的公平策略的锁。公平策略值为真，则确保等待时间最长的线程首先获得锁。取值为假的公平

策略将锁给任意一个在等待的线程。被多个线程访问的使用公正锁的程序，其整体性能可能比那些使用默认设置的程序差，但是在获取锁且避免资源缺乏时可以有更小的时间变化。

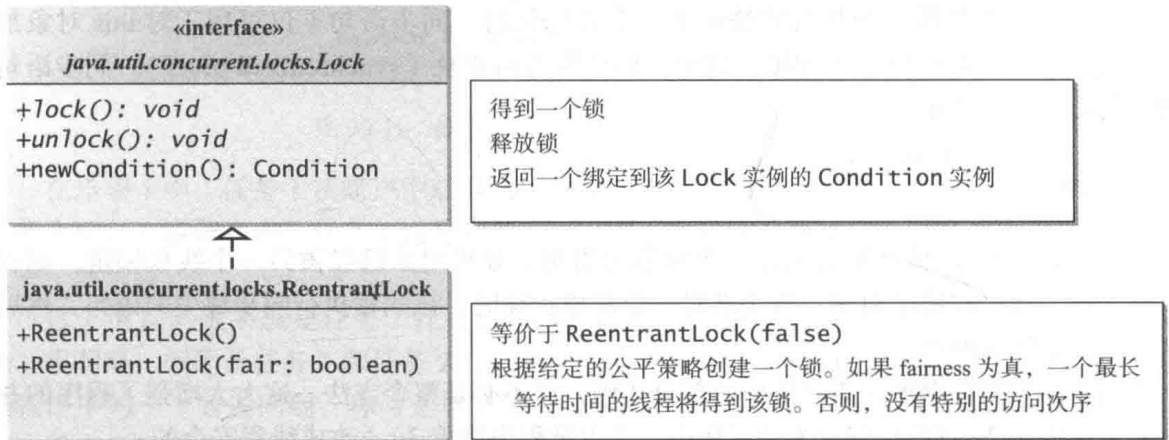


图 30-13 ReentrantLock 类实现接口 Lock 来表示一个锁

程序清单 30-5 修改程序清单 30-7，使用显式锁来同步账号的修改。

程序清单 30-5 AccountWithSyncUsingLock.java

```

1  import java.util.concurrent.*;
2  import java.util.concurrent.locks.*;
3
4  public class AccountWithSyncUsingLock {
5      private static Account account = new Account();
6
7      public static void main(String[] args) {
8          ExecutorService executor = Executors.newCachedThreadPool();
9
10         // Create and launch 100 threads
11         for (int i = 0; i < 100; i++) {
12             executor.execute(new AddAPennyTask());
13         }
14
15         executor.shutdown();
16
17         // Wait until all tasks are finished
18         while (!executor.isTerminated()) {
19             ;
20         }
21
22         System.out.println("What is balance? " + account.getBalance());
23     }
24
25     // A thread for adding a penny to the account
26     public static class AddAPennyTask implements Runnable {
27         public void run() {
28             account.deposit(1);
29         }
30     }
31
32     // An inner class for Account
33     public static class Account {
34         private static Lock lock = new ReentrantLock(); // Create a lock
35         private int balance = 0;
36
37         public int getBalance() {
38             ;
39         }
40     }
  
```



```
39
40 public void deposit(int amount) {
41     lock.lock(); // Acquire the lock
42
43     try {
44         int newBalance = balance + amount;
45
46         // This delay is deliberately added to magnify the
47         // data-corruption problem and make it easy to see.
48         Thread.sleep(5);
49
50         balance = newBalance;
51     }
52     catch (InterruptedException ex) {
53     }
54     finally {
55         lock.unlock(); // Release the lock
56     }
57 }
58 }
59 }
```

第 33 行创建一个锁，第 41 行获取该锁，第 55 行释放该锁。

提示：在对 `lock()` 的调用之后紧随一个 `try-catch` 块并且在 `finally` 子句中释放这个锁是一个很好的编程习惯，如第 41 ~ 56 行所示，这样可以确保锁被释放。

程序清单 30-5 可以为 `deposit` 使用同步方法来实现，而不是使用锁。通常，使用 `synchronized` 方法或语句比使用相互排斥的显式锁简单些。然而，使用显式锁对同步具有状态的线程更加直观和灵活，如下节所述。

✓ 复习题

30.18 如何创建一个锁对象？如何得到一个锁和释放一个锁？

30.9 线程间协作

要点提示：锁上的条件可以用于协调线程之间的交互。

通过保证在临界区上多个线程的相互排斥，线程同步完全可以避免竞争条件的发生，但是有时候，还需要线程之间的相互协作。可以使用条件实现线程间通信。一个线程可以指定在某种条件下该做什么。条件是通过调用 `Lock` 对象的 `newCondition()` 方法而创建的对象。一旦创建了条件，就可以使用 `await()`、`signal()` 和 `signalAll()` 方法来实现线程之间的相互通信，如图 30-14 所示。`await()` 方法可以让当前线程进入等待，直到条件发生。`signal()` 方法唤醒一个等待的线程，而 `signalAll()` 唤醒所有等待的线程。

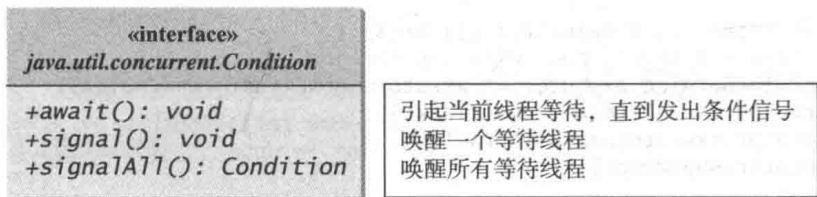


图 30-14 `Condition` 接口定义完成同步的方法

让我们用一个例子演示线程通信。假设创建并启动两个任务，一个用来向账户中存款，另一个从同一账户中提款。当提款的数额大于账户的当前余额时，提款线程必须等待。不管什么时候，只要向账户新存入一笔资金，存储线程必须通知提款线程重新尝试。如果余额仍

未达到提款的数额，提款线程必须继续等待新的存款。

为了同步这些操作，使用一个具有条件的锁 `newDeposit`（即增加到账户的新存款）。如果余额小于取款数额，提款任务将等待 `newDeposit` 条件。当存款任务给账户增加资金时，存款任务唤醒等待中的提款任务再次尝试。两个任务之间的交互如图 30-15 所示。

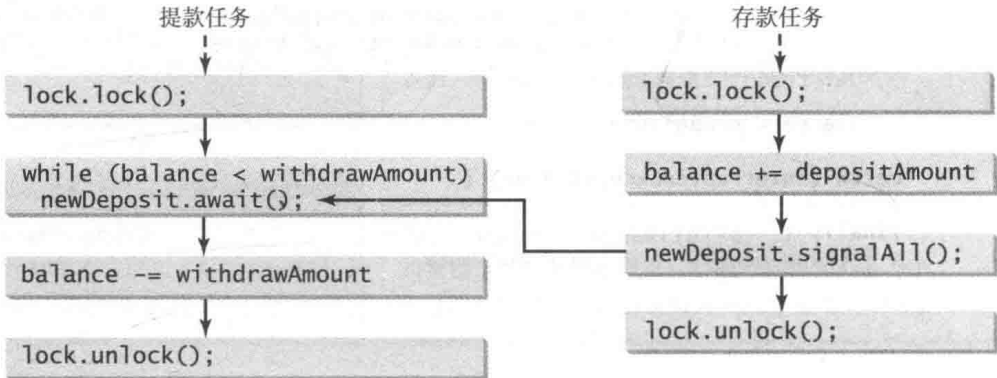


图 30-15 条件 `newDeposit` 用于两个线程间通信

从 `Lock` 对象中创建条件。为了使用条件，必须首先获取锁。`await()` 方法让线程等待并且自动释放条件上的锁。一旦条件正确，线程重新获取锁并且继续执行。

假设初始余额为 0，存入额和提取额是随机产生的。程序清单 30-6 给出程序。程序运行结果的一个示例如图 30-16 所示。

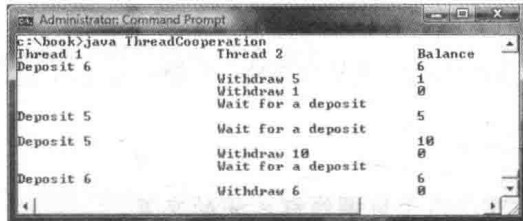


图 30-16 如果没有足够的金额供提取，则取款任务等待

程序清单 30-6 ThreadCooperation.java

```

1 import java.util.concurrent.*;
2 import java.util.concurrent.locks.*;
3
4 public class ThreadCooperation {
5     private static Account account = new Account();
6
7     public static void main(String[] args) {
8         // Create a thread pool with two threads
9         ExecutorService executor = Executors.newFixedThreadPool(2);
10        executor.execute(new DepositTask());
11        executor.execute(new WithdrawTask());
12        executor.shutdown();
13
14        System.out.println("Thread 1\t\tThread 2\t\tBalance");
15    }
16
17    public static class DepositTask implements Runnable {
18        @Override // Keep adding an amount to the account
19        public void run() {
20            try { // Purposely delay it to let the withdraw method proceed
  
```

```
21         while (true) {
22             account.deposit((int)(Math.random() * 10) + 1);
23             Thread.sleep(1000);
24         }
25     }
26     catch (InterruptedException ex) {
27         ex.printStackTrace();
28     }
29 }
30 }
31
32 public static class WithdrawTask implements Runnable {
33     @Override // Keep subtracting an amount from the account
34     public void run() {
35         while (true) {
36             account.withdraw((int)(Math.random() * 10) + 1);
37         }
38     }
39 }
40
41 // An inner class for account
42 private static class Account {
43     // Create a new lock
44     private static Lock lock = new ReentrantLock();
45
46     // Create a condition
47     private static Condition newDeposit = lock.newCondition();
48
49     private int balance = 0;
50
51     public int getBalance() {
52         return balance;
53     }
54
55     public void withdraw(int amount) {
56         lock.lock(); // Acquire the lock
57         try {
58             while (balance < amount) {
59                 System.out.println("\t\t\tWait for a deposit");
60                 newDeposit.await();
61             }
62
63             balance -= amount;
64             System.out.println("\t\t\tWithdraw " + amount +
65                 "\t\t" + getBalance());
66         }
67         catch (InterruptedException ex) {
68             ex.printStackTrace();
69         }
70         finally {
71             lock.unlock(); // Release the lock
72         }
73     }
74
75     public void deposit(int amount) {
76         lock.lock(); // Acquire the lock
77         try {
78             balance += amount;
79             System.out.println("Deposit " + amount +
80                 "\t\t\t\t\t" + getBalance());
81
82             // Signal thread waiting on the condition
83             newDeposit.signalAll();
84         }
```

```

85         finally {
86             lock.unlock(); // Release the lock
87         }
88     }
89 }
90 }

```

该示例创建一个名为 `Account` 的内部类来模拟账户，该类中包含两个方法：`deposit(int)` 和 `withdraw(int)`。创建一个名为 `DepositTask` 的类向账户中添加金额，创建一个名为 `WithdrawTask` 的类从余额中提取金额，再创建一个主类，用于创建并启动两个线程。

程序创建并提交存款任务（第 10 行）和提款任务（第 11 行）。为了让提款任务运行，特意让存款任务进入休眠状态（第 23 行）。如果没有足够的资金可提取，则提款任务等待（第 59 行）存款任务中余额变化的通知（第 83 行）。

第 44 行创建一个锁，锁上名为 `newDeposit` 的条件在第 47 行创建。一个条件对应一个锁。在等待和通知状态之前，线程必须先获取该条件的锁。当没有足够可取的数目时，提款任务在第 56 行获取锁，等待 `newDeposit` 条件（第 60 行），并且在第 71 行释放该锁。存款任务在第 76 行获取锁，在有新的钱存入之后通知所有 `newDeposit` 条件的等待线程（第 83 行）。

如果将第 58 ~ 61 行的 `while` 循环用下面的 `if` 语句代替，会出现什么情况？

```

if (balance < amount) {
    System.out.println("\t\t\tWait for a deposit");
    newDeposit.await();
}

```

只要余额发生变化，存款任务都会通知提款任务。当唤醒提款任务时，条件 (`balance < amount`) 的判断结果可能仍然为 `true`。如果使用 `if` 语句，提款任务有可能导致不正确的提款。如果使用循环语句，则提款任务可以有重新检验条件的机会。因此，在执行一个提款操作前应该在循环语句中测试条件。

【警告】 一旦线程调用条件上的 `await()`，线程就进入等待状态，等待恢复的信号。如果忘记对状态调用 `signal()` 或者 `signalAll()`，那么线程就永远等待下去。

【警告】 条件由 `Lock` 对象创建。为了调用它的方法（例如，`await()`、`signal()` 和 `signalAll()`），必须首先拥有锁。如果没有获取锁就调用这些方法，会抛出 `IllegalMonitorStateException` 异常。

锁和条件是 Java 5 中的新内容。在 Java 5 之前，线程通信是使用对象的内置监视器编程实现的。锁和条件比内置监视器更加强大且灵活，因此无须使用监视器。然而，如果使用遗留的 Java 代码，就可能会碰到 Java 的内置监视器。

监视器（monitor）是一个相互排斥且具备同步能力的对象。监视器中的一个时间点上，只能有一个线程执行一个方法。线程通过获取监视器上的锁进入监视器，并且通过释放锁退出监视器。任意对象都可能是一个监视器。一旦一个线程锁住对象，该对象就成为监视器。加锁是通过在方法或块上使用 `synchronized` 关键字来实现的。在执行同步方法或块之前，线程必须获取锁。如果条件不适合线程继续在监视器内执行，线程可能在监视器中等待。可以对监视器对象调用 `wait()` 方法来释放锁，这样其他的一些监视器中的线程就可以获取它，也就有可能改变监视器的状态。当条件合适时，另一线程可以调用 `notify()` 或 `notifyAll()` 方法来通知一个或所有的等待线程重新获取锁并且恢复执行。调用这些方法的模板如图 30-17 所示。

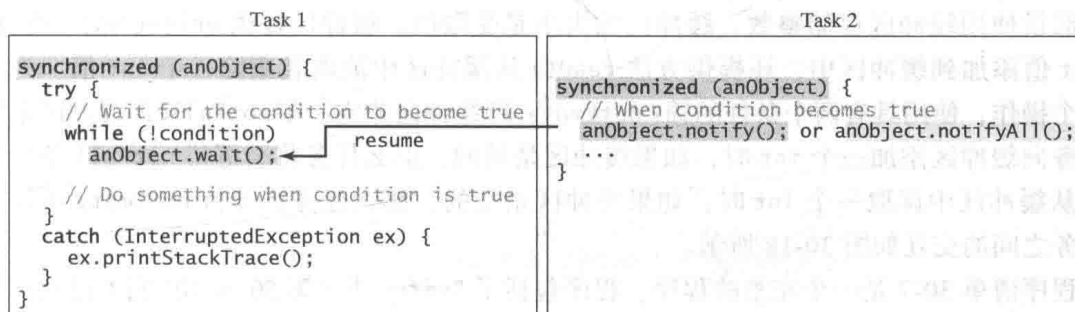


图 30-17 wait()、notify() 和 notifyAll() 方法协调线程间通信

wait()、notify() 和 notifyAll() 方法必须是在这些方法的接收对象的同步方法或同步块中调用。否则，就会出现 `IllegalMonitorStateException` 异常。

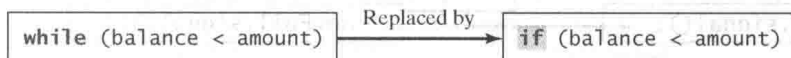
当调用 wait() 方法时，它终止线程同时释放对象的锁。当线程被通知之后重新启动时，锁就被重新自动获取。

对象上的 wait()、notify() 和 notifyAll() 方法类似于条件上的 await()、signal() 和 signalAll() 方法。

✓ 复习题

30.19 如何创建锁的条件？方法 await()、signal()、signalAll() 的用途分别是什么？

30.20 如果将程序清单 30-6 中第 58 行的 while 循环变成 if 语句，那么会发生什么？



30.21 为什么下面的类会有语法错误？

```

public class Test implements Runnable {
    public static void main(String[] args) {
        new Test();
    }
}

```

```

public Test() throws InterruptedException {
    Thread thread = new Thread(this);
    thread.sleep(1000);
}

```

```

public synchronized void run() {
}
}

```

30.22 什么是造成 `IllegalMonitorStateException` 异常的可能原因？

30.23 任意对象都能调用 wait()、notify() 和 notifyAll() 吗？这些方法的目的是什么？

30.24 下面的代码有什么错误？

```

synchronized (object1) {
    try {
        while (!condition) object2.wait();
    }
    catch (InterruptedException ex) {
    }
}

```

30.10 示例学习：生产者 / 消费者

🔑 要点提示：本节给出经典的生产者 / 消费者示例，来演示线程的协调。

假设使用缓冲区存储整数。缓冲区的大小是受限的。缓冲区提供 `write(int)` 方法将一个 `int` 值添加到缓冲区中，还提供方法 `read()` 从缓冲区中读取和删除一个 `int` 值。为了同步这个操作，使用具有两个条件的锁：`notEmpty`（即缓冲区非空）和 `notFull`（即缓冲区未满）。当任务向缓冲区添加一个 `int` 时，如果缓冲区是满的，那么任务将会等待 `notFull` 条件。当任务从缓冲区中读取一个 `int` 时，如果缓冲区是空的，那么任务将等待 `notEmpty` 条件。两个任务之间的交互如图 30-18 所示。

程序清单 30-7 是一个完整的程序。程序包括了 `Buffer` 类（第 50 ~ 101 行）以及重复向缓冲区产生数字和重复从缓冲区消耗数字的两个任务（第 16 ~ 47 行）。`write(int)` 方法（第 62 ~ 79 行）向缓冲区添加一个整数。`read()` 方法（第 81 ~ 100 行）从缓冲区删除和返回一个整数。

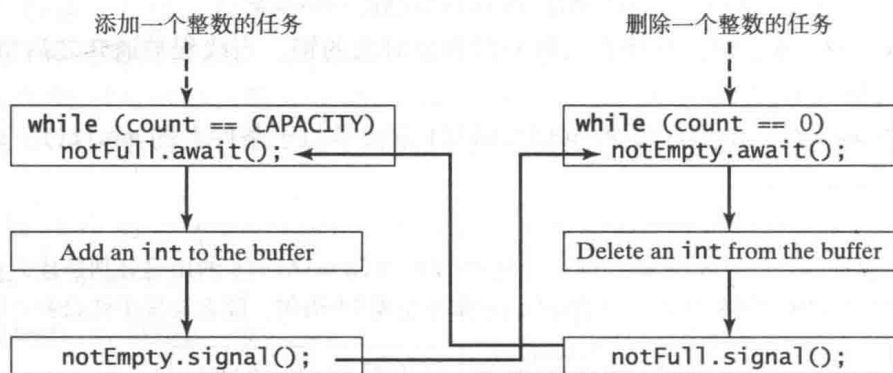


图 30-18 条件 `notFull` 和 `notEmpty` 用来协调任务交互

缓冲区实际上是一个先进先出的队列（第 52 ~ 53 行）。锁的条件 `notEmpty` 和 `notFull` 在第 59 ~ 60 行创建。条件和锁捆绑在一起。在应用一个条件之前必须获取一个锁。如果使用 `wait()` 和 `notify()` 方法重写这个例子，必须指派两个对象作为监视器。

程序清单 30-7 ConsumerProducer.java

```

1 import java.util.concurrent.*;
2 import java.util.concurrent.locks.*;
3
4 public class ConsumerProducer {
5     private static Buffer buffer = new Buffer();
6
7     public static void main(String[] args) {
8         // Create a thread pool with two threads
9         ExecutorService executor = Executors.newFixedThreadPool(2);
10        executor.execute(new ProducerTask());
11        executor.execute(new ConsumerTask());
12        executor.shutdown();
13    }
14
15    // A task for adding an int to the buffer
16    private static class ProducerTask implements Runnable {
17        public void run() {
18            try {
19                int i = 1;
20                while (true) {
21                    System.out.println("Producer writes " + i);
22                    buffer.write(i++); // Add a value to the buffer
23                    // Put the thread into sleep
24                    Thread.sleep((int)(Math.random() * 10000));

```

```

25     }
26 }
27 catch (InterruptedException ex) {
28     ex.printStackTrace();
29 }
30 }
31 }
32
33 // A task for reading and deleting an int from the buffer
34 private static class ConsumerTask implements Runnable {
35     public void run() {
36         try {
37             while (true) {
38                 System.out.println("\t\t\tConsumer reads " + buffer.read());
39                 // Put the thread into sleep
40                 Thread.sleep((int)(Math.random() * 10000));
41             }
42         }
43         catch (InterruptedException ex) {
44             ex.printStackTrace();
45         }
46     }
47 }
48
49 // An inner class for buffer
50 private static class Buffer {
51     private static final int CAPACITY = 1; // buffer size
52     private java.util.LinkedList<Integer> queue =
53         new java.util.LinkedList<>();
54
55     // Create a new lock
56     private static Lock lock = new ReentrantLock();
57
58     // Create two conditions
59     private static Condition notEmpty = lock.newCondition();
60     private static Condition notFull = lock.newCondition();
61
62     public void write(int value) {
63         lock.lock(); // Acquire the lock
64         try {
65             while (queue.size() == CAPACITY) {
66                 System.out.println("Wait for notFull condition");
67                 notFull.await();
68             }
69
70             queue.offer(value);
71             notEmpty.signal(); // Signal notEmpty condition
72         }
73         catch (InterruptedException ex) {
74             ex.printStackTrace();
75         }
76         finally {
77             lock.unlock(); // Release the lock
78         }
79     }
80
81     public int read() {
82         int value = 0;
83         lock.lock(); // Acquire the lock
84         try {
85             while (queue.isEmpty()) {
86                 System.out.println("\t\t\tWait for notEmpty condition");
87                 notEmpty.await();
88             }

```



```
89
90     value = queue.remove();
91     notFull.signal(); // Signal notFull condition
92 }
93 catch (InterruptedException ex) {
94     ex.printStackTrace();
95 }
96 finally {
97     lock.unlock(); // Release the lock
98     return value;
99 }
100 }
101 }
102 }
```

这个程序运行的示例如图 30-19 所示。

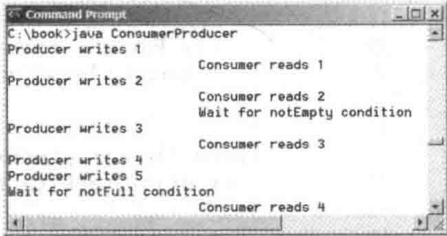


图 30-19 使用锁和条件实现生产者和消费者线程之间的通信

复习题

- 30.25 Buffer 类中的 read 和 write 方法可以并行执行吗?
- 30.26 调用 read 方法时, 如果队列为空会发生什么?
- 30.27 调用 write 方法时, 如果队列满了会发生什么?

30.11 阻塞队列

要点提示: Java 合集框架提供了 ArrayBlockingQueue、LinkedBlockingQueue 和 PriorityBlocking Queue 来支持阻塞队列。

20.9 节介绍了队列和优先队列。阻塞队列 (blocking queue) 在试图向一个满队列添加元素或者从空队列中删除元素时会导致线程阻塞。BlockingQueue 接口继承了 java.util.Queue, 并且提供同步的 put 和 take 方法向队列尾部添加元素, 以及从队列头部删除元素, 如图 30-20 所示。

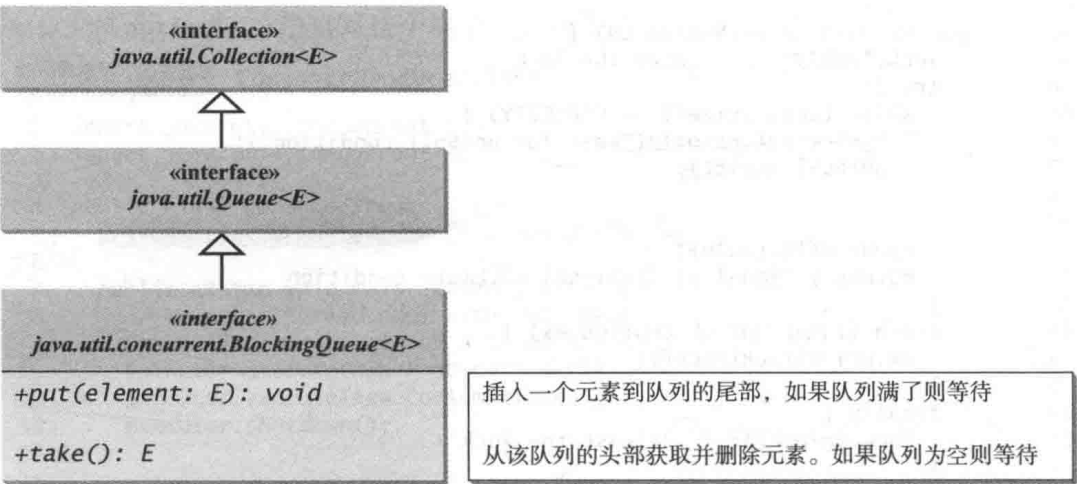


图 30-20 BlockingQueue 是 java.util.Queue 的子接口

Java 支持的三个具体的阻塞队列 ArrayBlockingQueue、LinkedBlockingQueue 和 PriorityBlockingQueue 如图 30-21 所示。它们都在 java.util.concurrent 包中。ArrayBlockingQueue 使用数组实现阻塞队列。必须指定一个容量或者可选的公平性策略来构造 ArrayBlockingQueue。LinkedBlockingQueue 使用链表实现阻塞队列。可以创建无边界的或有边界的 LinkedBlockingQueue。PriorityBlockingQueue 是优先队列。可以创建无边界的或

有边界的优先队列。

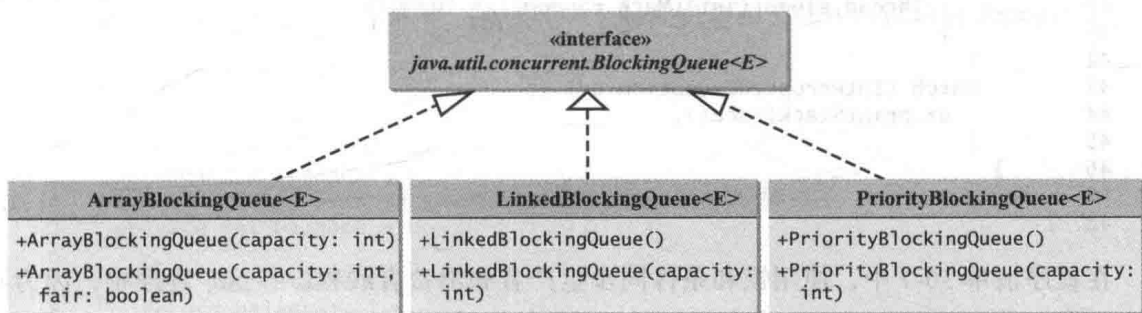


图 30-21 `ArrayBlockingQueue`、`LinkedBlockingQueue` 和 `PriorityBlockingQueue` 是具体的阻塞队列

{ } 注意：对于无边界的 `LinkedBlockingQueue` 或 `PriorityBlockingQueue` 而言，`put` 方法将永远不会阻塞。

程序清单 30-8 给出使用 `ArrayBlockingQueue` 来简化程序清单 30-10 中的消费者/生产者例子。第 5 行创建一个 `ArrayBlockingQueue` 来存储整数。生产者线程将一个整数放入队列中（第 22 行），而消费者线程从队列中取走一个整数（第 38 行）。

程序清单 30-8 `ConsumerProducerUsingBlockingQueue.java`

```

1  import java.util.concurrent.*;
2
3  public class ConsumerProducerUsingBlockingQueue {
4      private static ArrayBlockingQueue<Integer> buffer =
5          new ArrayBlockingQueue<>(2);
6
7      public static void main(String[] args) {
8          // Create a thread pool with two threads
9          ExecutorService executor = Executors.newFixedThreadPool(2);
10         executor.execute(new ProducerTask());
11         executor.execute(new ConsumerTask());
12         executor.shutdown();
13     }
14
15     // A task for adding an int to the buffer
16     private static class ProducerTask implements Runnable {
17         public void run() {
18             try {
19                 int i = 1;
20                 while (true) {
21                     System.out.println("Producer writes " + i);
22                     buffer.put(i++); // Add any value to the buffer, say, 1
23                     // Put the thread into sleep
24                     Thread.sleep((int)(Math.random() * 10000));
25                 }
26             } catch (InterruptedException ex) {
27                 ex.printStackTrace();
28             }
29         }
30     }
31
32     // A task for reading and deleting an int from the buffer
33     private static class ConsumerTask implements Runnable {
34         public void run() {
35             try {
36                 while (true) {
37                     System.out.println("\t\t\tConsumer reads " + buffer.take());

```

```
39         // Put the thread into sleep
40         Thread.sleep((int)(Math.random() * 10000));
41     }
42 }
43 catch (InterruptedException ex) {
44     ex.printStackTrace();
45 }
46 }
47 }
48 }
```

在程序清单 30-7 中，使用锁和条件同步生产者和消费者线程。在这个程序中，因为同步已经在 ArrayBlockingQueue 中实现，所以无需使用锁和条件。

复习题

- 30.28 什么是阻塞队列？Java 中支持什么阻塞队列？
- 30.29 使用什么方法来添加一个元素到 ArrayBlockingQueue 中？如果队列满了会发生什么？
- 30.30 使用什么方法从 ArrayBlockingQueue 中获取一个元素？如果队列为空将发生什么？

30.12 信号量

要点提示：可以使用信号量来限制访问一个共享资源的线程数。
计算机科学中，信号量指对共同资源进行访问控制的对象。在访问资源之前，线程必须从信号量获取许可。在访问完资源之后，这个线程必须将许可返回给信号量，如图 30-22 所示。

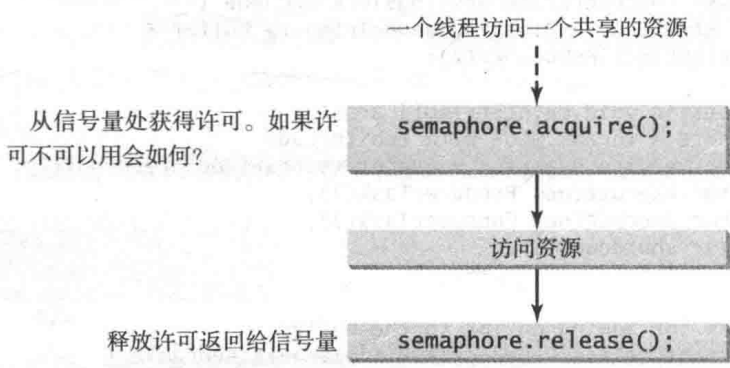


图 30-22 有限数量的线程可以访问受信号量控制的共享资源

为了创建信号量，必须确定许可的数量，同时可选用公平策略，如图 30-23 所示。任务通过调用信号量的 acquire() 方法来获得许可，通过调用信号量的 release() 方法来释放许可。一旦获得许可，信号量中可用许可的总数减 1。一旦许可被释放，信号量中可用许可的总数加 1。

java.util.concurrent.Semaphore	
+Semaphore(numberOfPermits: int)	创建一个具有指定数目的许可的信号量。公平性策略参数为假
+Semaphore(numberOfPermits: int, fair: boolean)	创建一个具有指定数目的许可以及公平性策略的信号量
+acquire(): void	从该信号量获取一个许可。如果许可不可用，线程将被阻塞，直到一个许可可用
+release(): void	释放一个许可返回给信号量

图 30-23 Semaphore 类包含访问信号量的方法

只有一个许可的信号量可以用来模拟一个相互排斥的锁。程序清单 30-9 使用信号量修改了程序清单 30-6 中的 Account 内部类，确保同一个时间只有一个线程可以访问 deposit 方法。

程序清单 30-9 New Account Inner Class

```
1 // An inner class for Account
2 private static class Account {
3     // Create a semaphore
4     private static Semaphore semaphore = new Semaphore(1);
5     private int balance = 0;
6
7     public int getBalance() {
8         return balance;
9     }
10
11    public void deposit(int amount) {
12        try {
13            semaphore.acquire(); // Acquire a permit
14            int newBalance = balance + amount;
15
16            // This delay is deliberately added to magnify the
17            // data-corruption problem and make it easy to see
18            Thread.sleep(5);
19
20            balance = newBalance;
21        }
22        catch (InterruptedException ex) {
23        }
24        finally {
25            semaphore.release(); // Release a permit
26        }
27    }
28 }
```

程序在第 4 行创建具有一个许可的信号量。当执行第 13 行的存款方法时，一个线程首先获得许可。在余额更新之后，线程在第 25 行释放该许可。总是将 release() 方法放到 finally 子句中是一个很好的习惯，这样可以确保即使发生异常也能最终释放该许可。

✓ 复习题

30.31 锁和信号量之间的相似之处和不同之处在什么地方？

30.32 如何创建一个允许 3 个并行线程的信号量？如何获取一个信号量？如何释放一个信号量？

30.13 避免死锁

🔑 **要点提示：**可以采用正确的资源排序来避免死锁。

有时两个或多个线程需要在几个共享对象上获取锁，这可能会导致死锁。也就是说，每个线程已经获取了其中一个对象上的锁，而且正在等待另一个对象上的锁。考虑有两个线程和两个对象的情形，如图 30-24 所示。线程 1 获取 object1 上的锁，而线程 2 获取 object2 上的锁。现在线程 1 等待 object2 上的锁，线程 2 等待 object1 上的锁。每个线程都在等待另一个线程释放它所需要的锁，结果导致两个线程都无法继续运行。

使用一种称为资源排序的简单技术可以轻易地避免死锁的发生。该技术是给每一个需要锁的对象指定一个顺序，确保每个线程都按这个顺序来获取锁。例如，在图 30-24 中，假设按 object1、object2 的顺序对两个对象排序。采用资源排序技术，线程 2 必须先获取 object1 上的锁，然后才能获取 object2 上的锁。一旦线程 1 获取了 object1 上的锁，线程 2 必须等待 object1 上的锁。所以，线程 1 就能获取 object2 上的锁，不会再发生死锁现象。

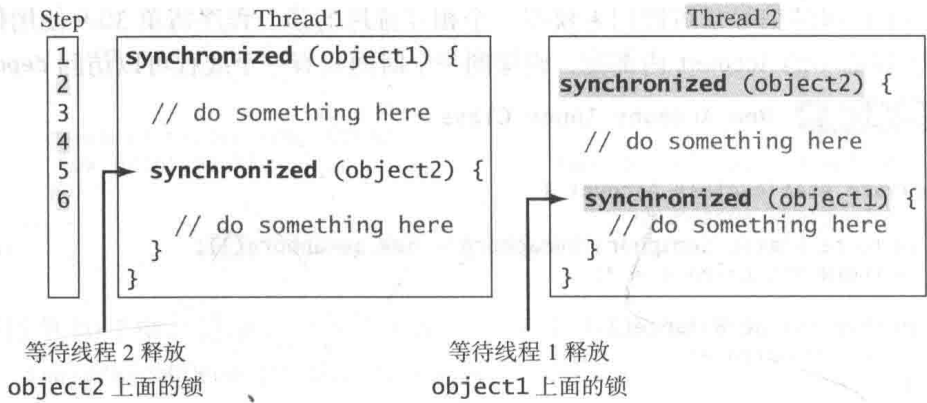


图 30-24 线程 1 和线程 2 是死锁的

复习题

30.33 什么是死锁？如何避免死锁？

30.14 线程状态

要点提示：线程状态可以表明一个线程的状态。

任务在线程中执行。线程可以是以下 5 种状态之一：新建、就绪、运行、阻塞或结束（如图 30-25 所示）。

新创建一个线程时，它就进入新建状态（New）。调用线程的 `start()` 方法启动线程后，它进入就绪状态（Ready）。就绪线程是可运行的，但可能还没有开始运行。操作系统必须为它分配 CPU 时间。

就绪线程开始运行时，它就进入运行状态。如果给定的 CPU 时间用完或调用线程的 `yield()` 方法，处于运行状态的线程可能就进入就绪状态。

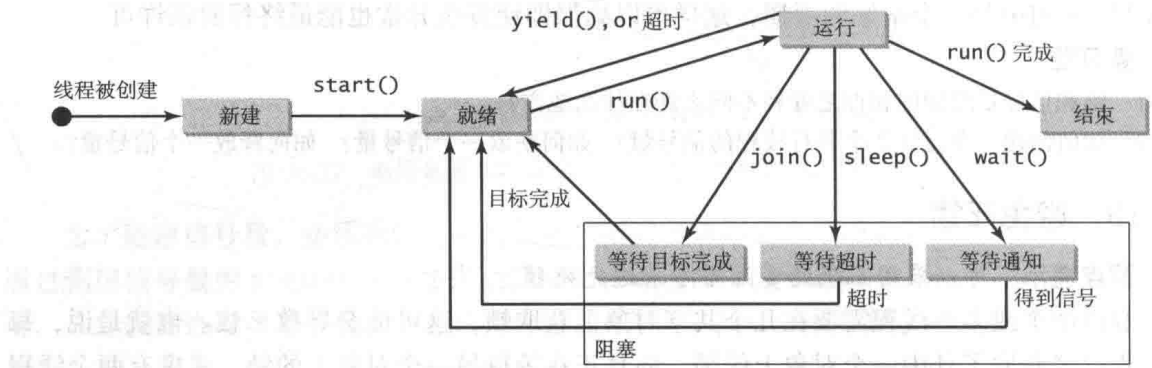


图 30-25 线程可以处于 5 种状态之一：新建、就绪、运行、阻塞或结束

有几种原因可能使线程进入阻塞状态（即非活动状态）。可能是它自己调用了 `join()`、`sleep()` 或 `wait()` 方法。它可能是在等待 I/O 操作的完成。当使得其处于非激活状态的动作不起作用时，阻塞线程可能被重新激活。例如，如果线程处于休眠状态并且休眠时间已过期，线程就会被重新激活并进入就绪状态。

最后，如果一个线程执行完它的 `run()` 方法，这个线程就被结束（finished）。
`isAlive()` 方法是用来判断线程状态的方法。如果线程处于就绪、阻塞或运行状态，则返回 `true`；如果线程处于新建并且没有启动的状态，或者已经结束，则返回 `false`。

方法 `interrupt()` 按下列方式中断一个线程：当线程当前处于就绪或运行状态时，给它设置一个中断标志；当线程处于阻塞状态时，它将被唤醒并进入就绪状态，同时抛出异常 `java.lang.InterruptedExceptio`。

✓ 复习题

30.34 什么是线程状态？描述一个线程的状态。

30.15 同步合集

🔑 要点提示：Java 合集框架为线性表、集合和映射表。

Java 合集框架中的类不是线程安全的；也就是说，如果它们同时被多个线程访问和更新，它们的内容可能被破坏。可以通过锁定合集或者同步合集来保护合集中的数据。

`Collections` 类提供 6 个静态方法来将合集转成同步版本，如图 30-26 所示。使用这些方法创建的合集称为同步包装类。

java.util.Collections	
+synchronizedCollection(c: Collection): Collection	返回一个同步合集
+synchronizedList(list: List): List	从一个给定的线性表返回一个同步线性表
+synchronizedMap(m: Map): Map	从一个给定的映射表返回一个同步映射表
+synchronizedSet(s: Set): Set	从一个给定的集合返回一个同步集合
+synchronizedSortedMap(s: SortedMap): SortedMap	从一个给定的排序映射表返回一个同步排序映射表
+synchronizedSortedSet(s: SortedSet): SortedSet	返回一个同步的排序集合

图 30-26 可以使用 `Collections` 类中的方法获得同步合集

调用 `synchronizedCollection(Collection c)` 会返回一个新的 `Collection` 对象，在里面所有访问和更新原来的合集 `c` 的方法都被同步。这些方法使用 `synchronized` 关键字来实现。例如，如下实现 `add` 方法：

```
public boolean add(E o) {  
    synchronized (this) {  
        return c.add(o);  
    }  
}
```

同步合集可以很安全地被多个线程并发地访问和修改。

❗ 注意：在 `java.util.Vector`、`java.util.Stack` 和 `java.util.Hashtable` 中的方法已经被同步。它们都是在 JDK1.0 中引入的旧类。从 JDK1.5 开始，应该使用 `java.util.ArrayList` 替换 `Vector`，用 `java.util.LinkedList` 替换 `Stack`，用 `java.util.Map` 替换 `Hashtable`。如果需要同步，就使用同步包装类。

这些同步包装类都是线程安全的，但是迭代器具有快速失效的特性。这就意味着当使用一个迭代器对一个合集进行遍历，而其依赖的合集被另一个线程修改时，那么迭代器会抛出异常 `java.util.ConcurrentModificationException` 报错，该异常是 `RuntimeException` 的一个子类。为了避免这个错误，需要创建一个同步合集对象，并且在遍历它时获取对象上的锁。例如，假设希望遍历一个集合，必须编写如下代码：

```
Set hashSet = Collections.synchronizedSet(new HashSet());
```

```
synchronized (hashSet) { // Must synchronize it  
    Iterator iterator = hashSet.iterator();
```



```

while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
}

```

不这样做可能会造成不确定的行为，例如 `ConcurrentModificationException`。

复习题

30.35 什么是同步集合？`ArrayList` 是同步的吗？如何使得其同步？

30.36 解释迭代器为什么会快速失效？

30.16 并行编程

要点提示：Fork/Join 框架用于在 Java 中实现并行编程。

多核系统的广泛应用产生了软件的革命。为了从多核系统受益，软件需要可以并行运行。JDK7 引入了新的 Fork/Join 框架用于并行编程，从而利用多核处理器。

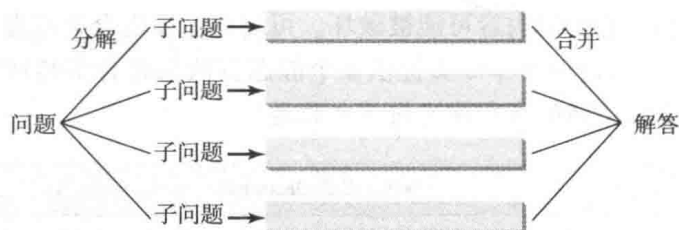


图 30-27 不重叠的子问题并行进行解决

Fork/Join 框架在图 30-27 中演示（图形很像一个分叉，从而得到这样的命名）。一个问题分为不重叠的子问题，这些子问题可以并行地独立解决。然后合并所有子问题的解答获得问题的整体解答。这是分而治之方法的并行实现。JDK7 的 Fork/Join 框架中，一个分解（fork）可以视为运行在一个线程上的独立任务。

框架使用 `ForkJoinTask` 类定义一个任务，如图 30-28 所示；同时，在一个 `ForkJoinPool` 的实例中执行一个任务，如图 30-29 所示。



图 30-28 `ForkJoinTask` 类定义了一个用于异步执行的任务

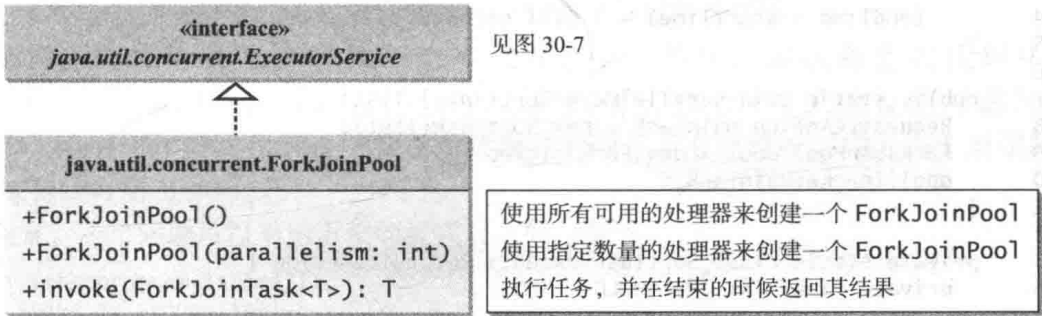


图 30-29 ForkJoinPool 执行 Fork/Join 任务

ForkJoinTask 是用于任务的抽象基类。一个 ForkJoinTask 是一个类似线程的实体，但是比普通的线程要轻量级得多，因为巨量的任务和子任务可以被 ForkJoinPool 中的少数真正的线程所执行。任务主要使用 fork() 和 join() 来协调。在一个任务上调用 fork() 会安排异步的执行，然后调用 join() 等待任务完成。invoke() 和 invokeAll(tasks) 方法都隐式地调用 fork() 来执行任务，以及 join() 等待任务完成，如果有结果则返回结果。注意，静态方法 invokeAll 使用 ... 语法来采用一个变长度的 ForkJoinTask 参数，这种做法在 7.9 节中介绍过。

Fork/Join 框架是设计用于并行的分而治之解决方案，分而治之本身是递归的。RecursiveAction 和 RecursiveTask 是 ForkJoinTask 的两个子类。要定义具体的任务类，类应该继承自 RecursiveAction 或者 RecursiveTask。RecursiveAction 用于不返回值的任务，而 RecursiveTask 用于返回值的任务。你自己的任务类应该重写 compute() 方法来指定任务是如何执行的。

现在我们采用合并排序来演示如何使用 Fork/Join 框架来开发并行程序。合并排序算法（在第 25.3 节中介绍）将数组分为两半，并且递归地对每一半都应用合并排序。当两部分排好序了，算法将它们合并。程序清单 30-10 给出了一个合并排序算法的并行实现，并将其执行时间与一个顺序的排序进行比较。

程序清单 30-10 ParallelMergeSort.java

```
1 import java.util.concurrent.RecursiveAction;
2 import java.util.concurrent.ForkJoinPool;
3
4 public class ParallelMergeSort {
5     public static void main(String[] args) {
6         final int SIZE = 7000000;
7         int[] list1 = new int[SIZE];
8         int[] list2 = new int[SIZE];
9
10        for (int i = 0; i < list1.length; i++)
11            list1[i] = list2[i] = (int)(Math.random() * 100000000);
12
13        long startTime = System.currentTimeMillis();
14        parallelMergeSort(list1); // Invoke parallel merge sort
15        long endTime = System.currentTimeMillis();
16        System.out.println("\nParallel time with "
17            + Runtime.getRuntime().availableProcessors() +
18            " processors is " + (endTime - startTime) + " milliseconds");
19
20        startTime = System.currentTimeMillis();
21        MergeSort.mergeSort(list2); // MergeSort is in Listing 23.5
22        endTime = System.currentTimeMillis();
23        System.out.println("\nSequential time is " +
```

```

24         (endTime - startTime) + " milliseconds");
25     }
26
27     public static void parallelMergeSort(int[] list) {
28         RecursiveAction mainTask = new SortTask(list);
29         ForkJoinPool pool = new ForkJoinPool();
30         pool.invoke(mainTask);
31     }
32
33     private static class SortTask extends RecursiveAction {
34         private final int THRESHOLD = 500;
35         private int[] list;
36
37         SortTask(int[] list) {
38             this.list = list;
39         }
40
41         @Override
42         protected void compute() {
43             if (list.length < THRESHOLD)
44                 java.util.Arrays.sort(list);
45             else {
46                 // Obtain the first half
47                 int[] firstHalf = new int[list.length / 2];
48                 System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
49
50                 // Obtain the second half
51                 int secondHalfLength = list.length - list.length / 2;
52                 int[] secondHalf = new int[secondHalfLength];
53                 System.arraycopy(list, list.length / 2,
54                     secondHalf, 0, secondHalfLength);
55
56                 // Recursively sort the two halves
57                 invokeAll(new SortTask(firstHalf),
58                     new SortTask(secondHalf));
59
60                 // Merge firstHalf with secondHalf into list
61                 MergeSort.merge(firstHalf, secondHalf, list);
62             }
63         }
64     }
65 }

```

Parallel time with 2 processors is 2829 milliseconds
 Sequential time is 4751 milliseconds

由于排序算法不返回值，我们定义一个继承自 `RecursiveAction` 的具体类 `ForkJoinTask` (第 33 ~ 46 行)。重写了 `compute` 方法来实现一个递归的合并排序 (第 42 ~ 63 行)。如果线性表比较小，采用顺序方式解决更加高效 (第 44 行)。对于一个大的线性表，将其分为两半 (第 47 ~ 54 行)。两半分别并行排序 (第 57 和 58 行)，然后进行合并 (第 61 行)。

程序创建一个主 `ForkJoinTask` (第 28 行)，一个 `ForkJoinPool` (第 29 行)，然后将该主任务放在 `ForkJoinPool` 中执行 (第 30 行)。`invoke` 方法在主任务执行完后将返回。

执行主任务时，任务分为子任务，并通过使用 `invokeAll` 方法来调用子任务 (第 57 和 58 行)。`invokeAll` 方法在所有子任务都完成后将返回。注意，每个子任务又进一步递归地分为更加小的任务。巨量的子任务可以在池中创建和执行。`Fork/Join` 框架高效地自动执行和协调所有的任务。

程序清单 23-5 中定义了 `MergeSort` 类。程序调用 `MergeSort.merge` 来合并两个排好序的子线性表 (第 61 行)。程序也调用 `MergeSort.mergeSort` (第 21 行) 来顺序地使用合并排序

来对一个线性表进行排序。可以看到并行排序比顺序排序要快很多。

注意, 初始化线性表的循环也可以并行化。然后, 应该避免在代码中使用 `Math.random()`, 因为它是同步执行的, 不可以并行执行 (参见编程练习题 30.12)。`parallelMergeSort` 方法仅对一个整型的数组进行排序, 不能修改它成为一个通用的方法 (参见编程练习题 30.13)。

通常, 一个问题可以采用下面的模式进行并行解决:

```
if (the program is small)
    solve it sequentially;
else {
    divide the problem into nonoverlapping subproblems;
    solve the subproblems concurrently;
    combine the results from subproblems to solve the whole problem;
}
```

程序清单 30-11 开发了一个并行方法, 用于在线性表中查找最大数。

程序清单 30-11 ParallelMax.java

```
1  import java.util.concurrent.*;
2
3  public class ParallelMax {
4      public static void main(String[] args) {
5          // Create a list
6          final int N = 9000000;
7          int[] list = new int[N];
8          for (int i = 0; i < list.length; i++)
9              list[i] = i;
10
11         long startTime = System.currentTimeMillis();
12         System.out.println("\nThe maximal number is " + max(list));
13         long endTime = System.currentTimeMillis();
14         System.out.println("The number of processors is " +
15             Runtime.getRuntime().availableProcessors());
16         System.out.println("Time is " + (endTime - startTime)
17             + " milliseconds");
18     }
19
20     public static int max(int[] list) {
21         RecursiveTask<Integer> task = new MaxTask(list, 0, list.length);
22         ForkJoinPool pool = new ForkJoinPool();
23         return pool.invoke(task);
24     }
25
26     private static class MaxTask extends RecursiveTask<Integer> {
27         private final static int THRESHOLD = 1000;
28         private int[] list;
29         private int low;
30         private int high;
31
32         public MaxTask(int[] list, int low, int high) {
33             this.list = list;
34             this.low = low;
35             this.high = high;
36         }
37
38         @Override
39         public Integer compute() {
40             if (high - low < THRESHOLD) {
41                 int max = list[0];
42                 for (int i = low; i < high; i++)
43                     if (list[i] > max)
```

```

44         max = list[i];
45     return new Integer(max);
46 }
47 else {
48     int mid = (low + high) / 2;
49     RecursiveTask<Integer> left = new MaxTask(list, low, mid);
50     RecursiveTask<Integer> right = new MaxTask(list, mid, high);
51
52     right.fork();
53     left.fork();
54     return new Integer(Math.max(left.join().intValue(),
55                                right.join().intValue()));
56 }
57 }
58 }
59 }

```

The maximal number is 8999999
 The number of processors is 2
 Time is 44 milliseconds

由于该算法返回一个整数，我们通过继承 `Recursive<Integer>` 为分解合并操作定义一个任务类（第 26 ~ 58 行）。重写 `compute` 方法返回 `list[low..high]` 中的最大元素（第 39 ~ 57 行）。如果线性表较小，采用顺序方式解决更加高效（第 40 ~ 46 行）。对于一个大的线性表，将其分为两半（第 48 ~ 50 行），任务 `left` 和 `right` 分别找到左半边和右半边的最大元素。在任务上调用 `fork()` 将使得任务被执行（第 52 和 53 行）。`join()` 方法等待任务执行完，然后返回结果（第 54 和 55 行）。

✓ 复习题

- 30.37 如何定义一个 `ForkJoinTask`? `RecursiveAction` 和 `RecursiveTask` 的区别是什么?
- 30.38 如何告诉系统来执行一个任务?
- 30.39 可以使用什么方法来测试一个任务是否已经完成?
- 30.40 如何创建一个 `ForkJoinPool`? 如何将一个任务放到一个 `ForkJoinPool` 中?

关键术语

condition (条件)	multithreading (多线程)
deadlock (死锁)	race condition (竞争状态)
fail-fast (快速失效)	semaphore (信号量)
fairness policy (公平策略)	synchronization wrapper (同步包装类)
Fork/Join Framework (Fork/Join 框架)	synchronized block (同步块)
lock (锁)	thread (线程)
monitor (监视器)	thread-safe (线程安全)

本章小结

1. 每个任务都是 `Runnable` 接口的实例。线程就是一个便于任务执行的对象。可以通过实现 `Runnable` 接口来定义任务类，通过使用 `Thread` 构造方法包住一个任务来创建线程。
2. 一个线程对象被创建之后，可以使用 `start()` 方法启动线程，可以使用 `sleep(long)` 方法将线程转入休眠状态，以便其他线程获得运行的机会。
3. 线程对象从来不会直接调用 `run` 方法。到了执行某个线程的时候，Java 虚拟机调用 `run` 方法。类必须覆盖 `run` 方法，告诉系统线程运行时将会做什么。

得到一个 `ConcurrentModificationException` 异常。

- *30.10 (使用同步合集) 使用同步解决前一个练习题中的问题, 使得第二个线程不抛出 `ConcurrentModificationException` 异常。

30.15 节

- *30.11 (演示死锁) 编写一个演示死锁的程序。

30.18 节

- *30.12 (并行数组初始化器) 使用 Fork/Join 框架实现下面的方法, 可以设置随机值给线性表。

```
public static void parallelAssignValues(double[] list)
```

编写一个测试程序, 创建一个具有 9 000 000 个元素的线性表, 调用 `parallelAssignValues` 来赋随机值给线性表。另外实现一个顺序算法, 并且比较两种方法执行的时间。注意, 如果使用 `Math.random()`, 并行代码的执行时间将比顺序代码的执行时间差, 因为 `Math.random()` 是同步的, 不能并行执行。为了解决这个问题, 创建一个 `Random` 对象, 用于赋随机值给一个小的线性表。

- 30.13 (通用的并行合并排序) 修改程序清单 30-10, 定义一个通用的并行合并算法方法, 如下:

```
public static <E extends Comparable<E>> void  
    parallelMergeSort(E[] list)
```

- *30.14 (并行快速排序) 实现下面方法, 可以并行地使用快速排序对一个线性表进行排序 (参见程序清单 23-7)。

```
public static void parallelQuickSort(int[] list)
```

编写一个测试程序, 使用该并行方法和一个顺序方法, 对一个大小为 9 000 000 的线性表的执行时间进行计时。

- *30.15 (并行求和) 使用 Fork/Join 实现以下方法, 对一个线性表求和。

```
public static double parallelSum(double[] list)
```

编写一个测试程序, 对一个大小为 9 000 000 的 `double` 值求和。

- *30.16 (并行的矩阵加法) 编程练习题 8.5 描述了如何执行矩阵的加法。假设你有一个多处理器, 因此可以加速矩阵加法计算。实现以下并行方法:

```
public static double[][] parallelAddMatrix(  
    double[][] a, double[][] b)
```

编写一个测试程序, 分别对使用并行方法和顺序方法来实现两个 2000×2000 的矩阵加法计时。

- *30.17 (并行的矩阵乘法) 编程练习题 7.6 描述了如何执行矩阵的乘法。假设你有一个多处理器, 因此可以加速矩阵乘法计算。实现以下并行方法:

```
public static double[][] parallelMultiplyMatrix(  
    double[][] a, double[][] b)
```

编写一个测试程序, 分别对使用并行方法和顺序方法来实现两个 2000×2000 的矩阵乘法计时。

- *30.18 (并行计算八皇后问题) 修改程序清单 22-11, 开发一个并行算法, 为八皇后问题找到所有的解决方案。(提示: 运行 8 个子任务, 每个子任务将皇后放在第一行的不同列中。)

综合

- ***30.19 (排序动画) 为选择排序、插入排序和冒泡排序编写一个动画, 如图 30-31 所示。创建一个由整数 1, 2, ..., 50 构成的数组, 然后随机地打乱它。创建一个面板来显示这个数组。需要在每个单独的线程中调用每个排序方法。每个算法使用两个嵌套的循环。当算法结束外层循环的一

次遍历，让线程休眠 0.5 秒，然后重新显示该柱状图中的数组。将排好序的子数组的最后一个柱条彩色显示。

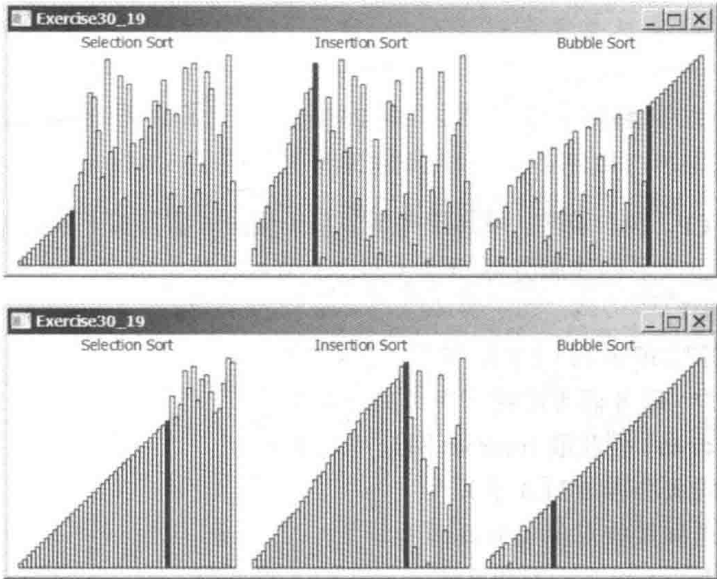


图 30-31 动画演示三种排序算法

***30.20（数独搜索模拟）修改编程练习题 22.21，显示搜索的中间结果。图 30-32 给出了动画的一个截屏，数字 2 放在如图 30-32a 所示的单元中，数字 3 放在如图 30-32b 所示的单元中，数字 1 放在如图 30-32c 所示的单元中。模拟显示所有的搜索步骤。

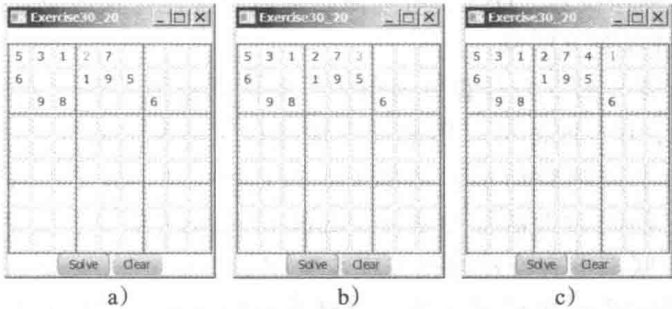


图 30-32 为数独问题动画显示中间搜索步骤

30.21（结合碰撞的弹球）修改编程练习题 20.5，使用一个线程来动画模拟弹球的移动。
***30.22（八皇后问题动画）修改程序清单 22-11，显示搜索的中间结果。如图 30-33 所示，高亮显示被搜索的当前行。每秒钟，棋盘的一个新状态被显示。

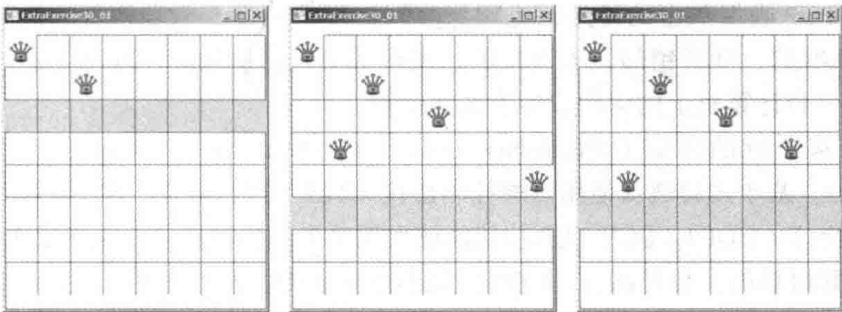


图 30-33 为八皇后问题动画显示中间搜索步骤

网 络

教学目标

- 解释术语：TCP、IP、域名、域名服务器、基于流的通信和基于数据包的通信（31.2 节）。
- 使用服务器套接字创建服务器程序（31.2.1 节）以及使用客户端套接字创建客户端程序（31.2.2 节）。
- 使用流套接字实现 Java 网络程序（31.2.3 节）。
- 开发一个客户 / 服务器程序的示例（31.2.4 节）。
- 使用 `InetAddress` 类获取 Internet 网址（31.3 节）。
- 开发多客户的服务器（31.4 节）。
- 在网络上发送和接收对象（31.5 节）。
- 开发一个在 Internet 上玩的交互式井字游戏（31.5 节）。

31.1 引言

 **要点提示：**计算机网络被用于在 Internet 上的计算机之间发送和接收消息。

为浏览网页或者发送邮件，计算机必须连接到互联网。互联网是连接数百万计算机的全球网络。计算机可以使用拨号、DSL、电缆调制解调器通过互联网服务提供商（ISP），或通过局域网（LAN）来连接到互联网。

当一台计算机需要与另一台计算机通信时，需要知道另一台计算机的地址。互联网协议（Internet Protocol, IP）地址可以用来唯一地标识互联网上的计算机。IP 地址由 4 段用点隔开的 0 ~ 255 的十进制数组成，例如 130.254.204.31。由于不容易记住这么多的数字，所以，经常将它们映射为被称为域名（domain name）的有含义的名字，例如 `liang.armstrong.edu`。在互联网上有特殊的称为域名服务器（domain name server, DNS）的服务器，它把主机的名字转换成 IP 地址。当一台计算机要连接 `liang.armstrong.edu` 时，它首先请求 DNS 将这个域名转换成 IP 地址，然后用这个 IP 地址来发送请求。

互联网协议是在互联网中从一台计算机向另一台计算机以包的形式传输数据的一种低层协议。两个和 IP 一起使用的较高层的协议是传输控制协议（transmission control protocol, TCP）和用户数据报协议（user datagram protocol, UDP）。TCP 能够让两台主机建立连接并交换数据流。TCP 确保数据的传送，也确保数据包以它们发送的顺序传送。UDP 是一种用在 IP 之上的标准的、低开销的、无连接的、主机对主机的协议。UDP 允许一台计算机上的应用程序向另一台计算机上的应用程序发送数据报。

Java 支持基于流的通信（stream-based communication）和基于包的通信（packet-based communication）。基于流的通信使用传输控制协议（TCP）进行数据传输，而基于包的通信使用用户数据报协议（UDP）。因为 TCP 协议能够发现丢失的传输信息并重新发送，所以，传输过程是无损的和可靠的。相对而言，UDP 协议不能保证传输没有丢失。因此，大多数 Java 程序设计采用基于流的通信，这也是本章的重点。基于包的通信在补充材料 III.P 中介绍。

31.2 客户端 / 服务器计算

要点提示：Java 提供 `ServerSocket` 类来创建服务器套接字，`Socket` 类来创建客户端套接字。Internet 上的两个程序通过使用 I/O 流的服务器套接字和客户端套接字进行通信。

网络功能紧密地集成在 Java 中。Java API 提供用于创建套接字的类来便于程序通过 Internet 通信。套接字 (socket) 是两台主机之间逻辑连接的端点，可以用来发送和接收数据。Java 对套接字通信的处理非常类似于对输入输出操作的处理，因此，程序对套接字读写就像对文件读写一样容易。

网络程序设计通常涉及一个服务器和一个或多个客户端。客户端向服务器发送请求，而服务器响应请求。客户端从尝试建立与服务器的连接开始，服务器可能接受或拒绝这个连接。一旦建立连接，客户端和服务器就可以通过套接字进行通信。

当客户端尝试连接到服务器时，服务器必须正在运行。服务器等待来自客户端的连接请求。创建服务器和客户端所需的语句如图 31-1 所示。

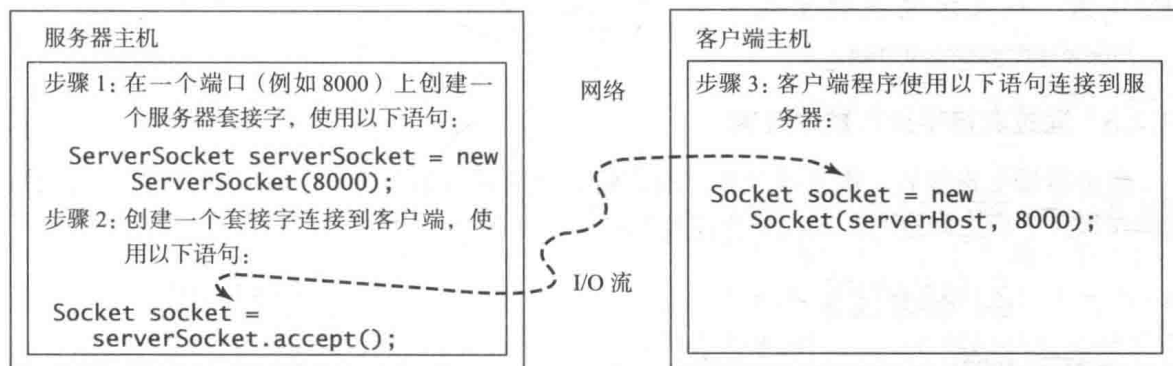


图 31-1 服务器创建一个服务器套接字，一旦建立起与客户端的连接，服务器就利用客户端套接字连接客户端

31.2.1 服务器套接字

要创建服务器，需要创建一个服务器套接字 (server socket)，并把它附加到一个端口上，服务器从这个端口监听连接。端口标识套接字上的 TCP 服务。端口号的范围为 0 ~ 65 536，但是 0 ~ 1024 是为特定服务保留的端口号。举例来说，电子邮件服务器运行在端口 25 上，Web 服务器通常运行在端口 80 上。可以选择任意一个当前没有被其他进程使用的端口。下面的语句创建一个服务器套接字 `serverSocket`：

```
ServerSocket serverSocket = new ServerSocket(port);
```

注意：如果试图在已经使用的端口上创建服务器套接字，就会导致 `java.net.BindException` 异常。

31.2.2 客户端套接字

创建服务器套接字之后，服务器可以使用下面的语句监听连接：

```
Socket socket = serverSocket.accept();
```

这个语句会一直等待，直到一个客户端连接到服务器套接字。客户端执行下面的语句，请求与服务器进行连接：

```
Socket socket = new Socket(serverName, port);
```

这条语句打开一个套接字，使得客户端程序能够与服务器进行通信。其中 `serverName` 是服务器的互联网主机名或 IP 地址。下面的语句在客户机的端口 8000 处创建一个套接字，用来连接到主机 130.254.204.33：

```
Socket socket = new Socket("130.254.204.33", 8000)
```

另一种做法是，使用域名创建套接字，如下所示：

```
Socket socket = new Socket("liang.armstrong.edu", 8000);
```

当使用主机名创建套接字时，Java 虚拟机要求 DNS 将主机名译成 IP 地址。

注意：程序可以使用主机名 `localhost` 或者 IP 地址 `127.0.0.1` 来引用客户端所运行的计算机。

注意：如果不能找到主机的话，Socket 构造方法就会抛出一个异常 `java.net.UnknownHostException`。

31.2.3 通过套接字进行数据传输

服务器接受连接后，服务器和客户端之间的通信就像输入输出（I/O）流一样进行操作。创建流以及它们之间进行数据交换所需要的语句，如图 31-2 所示。

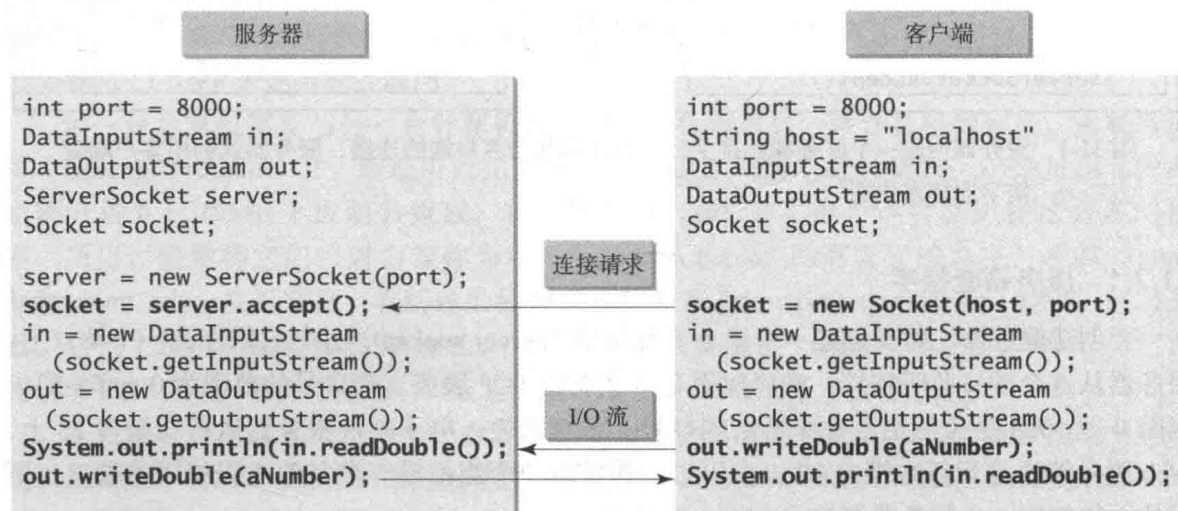


图 31-2 服务器与客户端在套接字上通过输入输出流进行数据交换

为了获得输入流和输出流，对套接字对象使用 `getInputStream()` 方法和 `getOutputStream()` 方法。例如，下面的语句从套接字创建一个称为 `input` 的 `InputStream` 流和一个称为 `output` 的 `OutputStream` 流：

```
InputStream input = socket.getInputStream();
OutputStream output = socket.getOutputStream();
```

`InputStream` 流和 `OutputStream` 流分别用来读取和写入字节。可以使用 `DataInputStream`、`DataOutputStream`、`BufferedReader` 和 `PrintWriter` 来包装 `InputStream` 和 `OutputStream`，

以读写像 `int`、`double` 或 `String` 之类的数据。例如，在下面的语句中，创建一个 `DataInputStream` 流 `input`，以及一个 `DataOutputStream` 流 `output`，用它们读取和写入基本数据类型的值：

```
DataInputStream input = new DataInputStream  
(socket.getInputStream());  
DataOutputStream output = new DataOutputStream  
(socket.getOutputStream());
```

服务器能够使用 `input.readDouble()` 方法从客户端接收 `double` 型数据，使用 `output.writeDouble(d)` 方法向客户端发送 `double` 型数据 `d`。

提示：由于文本 I/O 需要编码和解码，所以，二进制 I/O 的效率比文本 I/O 的效率更高。因此，最好使用二进制 I/O 在服务器和客户端之间进行数据传输，以便提高效率。

31.2.4 客户端 / 服务器示例

本例给出一个客户端程序和一个服务器程序。客户端向服务器发送数据。服务器接收数据，并用它来计算生成一个结果，然后，将这个结果返回给客户端。客户端在控制台上显示结果。在本例中，客户端发送的数据是圆的半径，服务器生成的结果是圆的面积（如图 31-3 所示）。

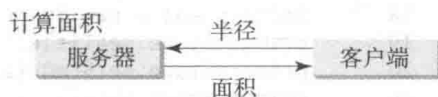


图 31-3 客户端将半径发送给服务器，服务器计算面积并将其发送给客户端

客户端通过输出流套接字的 `DataOutputStream` 发送半径，服务器通过输入流套接字的 `DataInputStream` 接收半径，如图 31-4a 所示。服务器计算面积，然后，通过输出流套接字的 `DataOutputStream` 把它发送给客户端，客户端通过输入流套接字的 `DataInputStream` 接收面积，如图 31-4b 所示。服务器程序和客户端程序在程序清单 31-1 和程序清单 31-2 中给出。图 31-5 给出服务器和客户端运行示例。

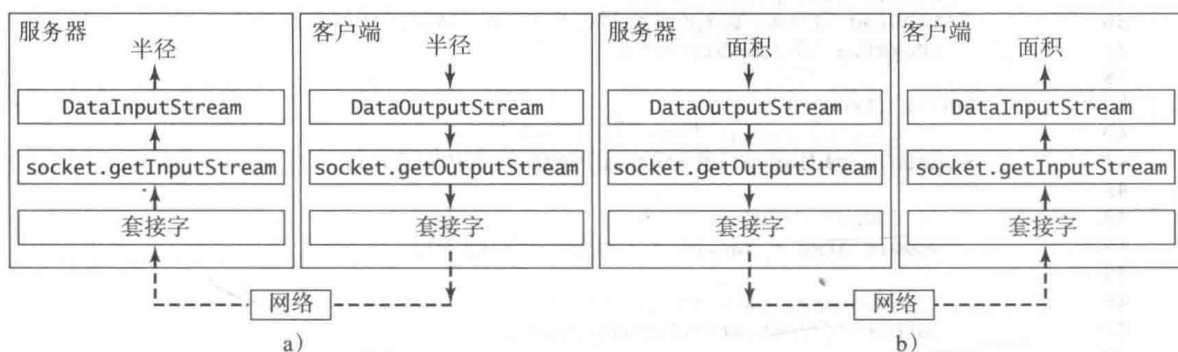


图 31-4 a) 客户端向服务器发送半径；b) 服务器向客户端发送面积

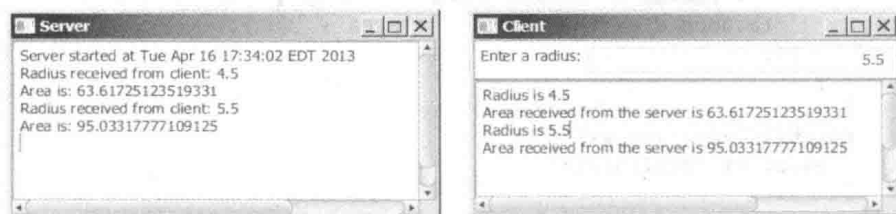


图 31-5 客户端将半径发送给服务器，服务器接收半径，计算面积，然后把面积发送给客户端

程序清单 31-1 Server.java

```
1  import java.io.*;
2  import java.net.*;
3  import java.util.Date;
4  import javafx.application.Application;
5  import javafx.application.Platform;
6  import javafx.scene.Scene;
7  import javafx.scene.control.ScrollPane;
8  import javafx.scene.control.TextArea;
9  import javafx.stage.Stage;
10
11 public class Server extends Application {
12     @Override // Override the start method in the Application class
13     public void start(Stage primaryStage) {
14         // Text area for displaying contents
15         TextArea ta = new TextArea();
16
17         // Create a scene and place it in the stage
18         Scene scene = new Scene(new ScrollPane(ta), 450, 200);
19         primaryStage.setTitle("Server"); // Set the stage title
20         primaryStage.setScene(scene); // Place the scene in the stage
21         primaryStage.show(); // Display the stage
22
23         new Thread(() -> {
24             try {
25                 // Create a server socket
26                 ServerSocket serverSocket = new ServerSocket(8000);
27                 Platform.runLater(() ->
28                     ta.appendText("Server started at " + new Date() + '\n'));
29
30                 // Listen for a connection request
31                 Socket socket = serverSocket.accept();
32
33                 // Create data input and output streams
34                 DataInputStream inputFromClient = new DataInputStream(
35                     socket.getInputStream());
36                 DataOutputStream outputToClient = new DataOutputStream(
37                     socket.getOutputStream());
38
39                 while (true) {
40                     // Receive radius from the client
41                     double radius = inputFromClient.readDouble();
42
43                     // Compute area
44                     double area = radius * radius * Math.PI;
45
46                     // Send area back to the client
47                     outputToClient.writeDouble(area);
48
49                     Platform.runLater(() -> {
50                         ta.appendText("Radius received from client: "
51                             + radius + '\n');
52                         ta.appendText("Area is: " + area + '\n');
53                     });
54                 }
55             } catch (IOException ex) {
56                 ex.printStackTrace();
57             }
58         }).start();
59     }
60 }
61 }
```

程序清单 31-2 Client.java

```
1  import java.io.*;
2  import java.net.*;
3  import javafx.application.Application;
4  import javafx.geometry.Insets;
5  import javafx.geometry.Pos;
6  import javafx.scene.Scene;
7  import javafx.scene.control.Label;
8  import javafx.scene.control.ScrollPane;
9  import javafx.scene.control.TextArea;
10 import javafx.scene.control.TextField;
11 import javafx.scene.layout.BorderPane;
12 import javafx.stage.Stage;
13
14 public class Client extends Application {
15     // IO streams
16     DataOutputStream toServer = null;
17     DataInputStream fromServer = null;
18
19     @Override // Override the start method in the Application class
20     public void start(Stage primaryStage) {
21         // Panel p to hold the label and text field
22         BorderPane paneForTextField = new BorderPane();
23         paneForTextField.setPadding(new Insets(5, 5, 5, 5));
24         paneForTextField.setStyle("-fx-border-color: green");
25         paneForTextField.setLeft(new Label("Enter a radius: "));
26
27         TextField tf = new TextField();
28         tf.setAlignment(Pos.BOTTOM_RIGHT);
29         paneForTextField.setCenter(tf);
30
31         BorderPane mainPane = new BorderPane();
32         // Text area to display contents
33         TextArea ta = new TextArea();
34         mainPane.setCenter(new ScrollPane(ta));
35         mainPane.setTop(paneForTextField);
36
37         // Create a scene and place it in the stage
38         Scene scene = new Scene(mainPane, 450, 200);
39         primaryStage.setTitle("Client"); // Set the stage title
40         primaryStage.setScene(scene); // Place the scene in the stage
41         primaryStage.show(); // Display the stage
42
43         tf.setOnAction(e -> {
44             try {
45                 // Get the radius from the text field
46                 double radius = Double.parseDouble(tf.getText().trim());
47
48                 // Send the radius to the server
49                 toServer.writeDouble(radius);
50                 toServer.flush();
51
52                 // Get area from the server
53                 double area = fromServer.readDouble();
54
55                 // Display to the text area
56                 ta.appendText("Radius is " + radius + "\n");
57                 ta.appendText("Area received from the server is "
58                     + area + '\n');
59             }
60             catch (IOException ex) {
61                 System.err.println(ex);
62             }
63         });
64     }
65 }
```

```
64
65     try {
66         // Create a socket to connect to the server
67         Socket socket = new Socket("localhost", 8000);
68         // Socket socket = new Socket("130.254.204.36", 8000);
69         // Socket socket = new Socket("drake.Armstrong.edu", 8000);
70
71         // Create an input stream to receive data from the server
72         fromServer = new DataInputStream(socket.getInputStream());
73
74         // Create an output stream to send data to the server
75         toServer = new DataOutputStream(socket.getOutputStream());
76     }
77     catch (IOException ex) {
78         ta.appendText(ex.toString() + '\n');
79     }
80 }
81 }
```

首先启动服务器程序，然后启动客户端程序。在客户端程序中，在文本域中输入一个半径，然后，按回车键将半径发送给服务器。服务器计算面积，再将它发回客户端。这个过程不断重复，直到两个程序中有一个结束。

有关网络的类都存放在包 `java.net` 中。当编写 Java 网络程序时，应该将该包导入。

执行下面的语句（`Server.java` 中的第 26 行），类 `Server` 创建一个 `ServerSocket` `serverSocket`，并把它附加到端口 8000 上：

```
ServerSocket serverSocket = new ServerSocket(8000);
```

然后服务器执行如下的语句（`Server.java` 中的第 31 行），开始启动对连接请求的监听：

```
Socket socket = serverSocket.accept();
```

服务器一直等待，直到客户端请求连接。在连接之后，服务器通过输入流从客户端读取半径，计算面积，然后通过输出流将结果发送给客户端。`ServerSocket accept()` 方法执行的时候花费时间。在 JavaFX 应用程序线程中运行该方法不合适。因此，将其放在一个单独的线程中（第 23 ~ 59 行）。更新 GUI 的语句需要使用 `Platform.runLater` 方法从 JavaFX 应用程序线程中运行（第 27 ~ 28 行，第 49 ~ 53 行）。

客户类 `Client` 使用下面的语句创建一个套接字，通过该套接字向同一台机器（`localhost`）上端口 8000 处的服务器请求连接（`Client.java` 中的第 67 行）：

```
Socket socket = new Socket("localhost", 8000);
```

如果在不同的机器上运行服务器和客户端，就应该将 `localhost` 替换为服务器的主机名或 IP 地址。在本例中，服务器和客户端运行在同一台机器上。

如果服务器没有运行，客户端程序将会因为异常 `java.net.ConnectException` 而终止。建立连接之后，为了接收服务器的数据和发送数据到服务器，客户端得到通过数据输入输出流包装的输入流和输出流。

如果启动服务器的时候收到一个 `java.net.BindException` 异常，说明服务器的端口正被占用。需要结束正在使用服务器该端口的进程，然后重新启动服务器。

注意：当创建一个服务器套接字时，必须为其指定一个端口（例如 8000）。当客户端与服务器相连（`Client.java` 的第 67 行）时，在客户端上创建一个套接字。这个套接字有它自己的本地端口。端口个数（例如 2047）由 Java 虚拟机自动选取，如图 31-6 所示。

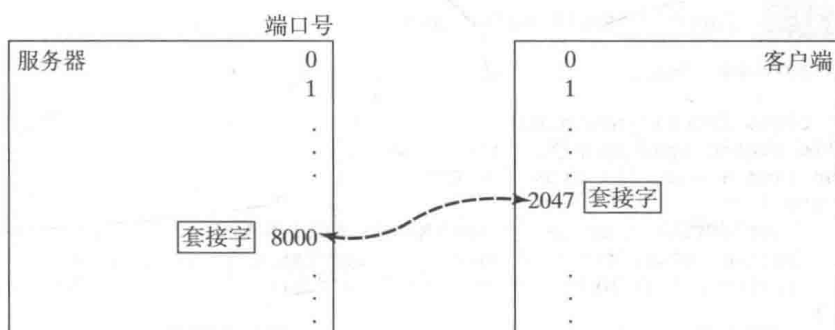


图 31-6 Java 虚拟机自动选择可用的端口为客户端创建套接字

为了看到客户端的本地端口，在 Client.java 中的第 70 行插入下面的语句：

```
System.out.println("local port: " + socket.getLocalPort());
```

复习题

- 31.1 如何创建服务器套接字？什么端口号是可用的？如果请求的端口号已经在使用，会发生什么现象？一个端口能与多个客户端连接吗？
- 31.2 服务器套接字和客户端套接字之间有什么区别？
- 31.3 客户端程序如何初始化一个连接？
- 31.4 服务器怎样接受连接请求？
- 31.5 数据是如何在客户端和服务器之间传输的？

31.3 InetAddress 类

要点提示：服务器程序可以使用 InetAddress 类来获得客户端的 IP 地址和主机名字等信息。

有时候，你可能想知道哪些人正连接在服务器上。这时可以使用类 InetAddress 来获取客户端的主机名和 IP 地址。InetAddress 类对 IP 地址建模。在服务器程序中使用下面的语句可以得到与客户端相连的套接字上的一个 InetAddress 实例：

```
InetAddress inetAddress = socket.getInetAddress();
```

然后，就可以显示客户端的主机名和 IP 地址，如下所示：

```
System.out.println("Client's host name is " +  
    inetAddress.getHostName());
```

```
System.out.println("Client's IP Address is " +  
    inetAddress.getHostAddress());
```

还可以使用静态方法 `getByName` 通过主机名或 IP 地址创建一个 InetAddress 的实例。例如，下面的语句为主机 `liang.armstrong.edu` 创建一个 InetAddress 实例：

```
InetAddress address = InetAddress.getByName("liang.armstrong.edu");
```

程序清单 31-3 给出了一个程序，这个程序标识从命令行传入的主机名和 IP 地址的参数。第 7 行使用 `getByName` 方法创建一个 InetAddress。第 8 行和第 9 行使用 `getHostName` 和 `getHostAddress` 方法来获得主机名和 IP 地址。图 31-7 给出这个程序的一个运行示例。

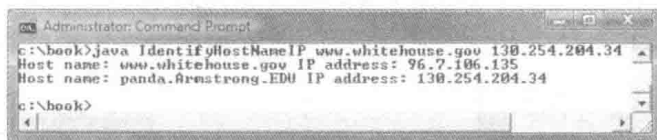


图 31-7 程序识别主机名和 IP 地址

程序清单 31-3 IdentifyHostNameIP.java

```

1  import java.net.*;
2
3  public class IdentifyHostNameIP {
4      public static void main(String[] args) {
5          for (int i = 0; i < args.length; i++) {
6              try {
7                  InetAddress address = InetAddress.getByName(args[i]);
8                  System.out.print("Host name: " + address.getHostName() + " ");
9                  System.out.println("IP address: " + address.getHostAddress());
10             }
11             catch (UnknownHostException ex) {
12                 System.err.println("Unknown host or IP address " + args[i]);
13             }
14         }
15     }
16 }

```

✓ 复习题

31.6 如何获得 `InetAddress` 的一个实例?

31.7 使用什么方法可以从一个 `InetAddress` 得到 IP 地址和主机名字?

31.4 服务多个客户

要点提示：一个服务器可以为多个客户端提供服务。对每个客户端的连接可以由一个线程来处理。

多个客户端同时连接到单个服务器是非常常见的。典型的情形是，一个服务器程序连续不停地在服务器计算机上运行，Internet 上各处的客户端都可以连接到它。可以使用线程处理服务器上多个客户端的同时访问。可以简单地为每个连接创建一个线程。下面给出服务器如何处理连接：

```

while (true) {
    Socket socket = serverSocket.accept(); // Connect to a client
    Thread thread = new ThreadClass(socket);
    thread.start();
}

```

服务器套接字可以有多个连接。while 循环的每次迭代创建一个新的连接。无论何时，只要建立一个新的连接，就创建一个新线程来处理服务器和新客户端之间的通信，这样，就可以有多个连接同时运行。

程序清单 31-4 创建一个服务器类为多个客户端同时提供服务。对于每个连接，服务器启动一个新线程。这个线程连续地接收来自客户端的输入（圆的半径），并把结果（圆的面积）发送回客户端（如图 31-8 所示）。客户端程序与程序清单 31-2 相同。图 31-9 给出一个服务器与两个客户端的运行示例。

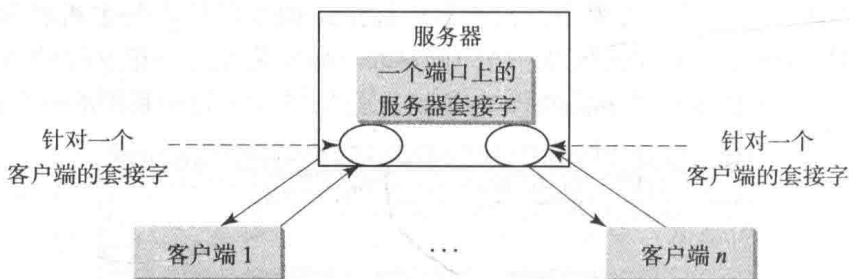


图 31-8 多线程可以使一个服务器处理多个独立的客户端

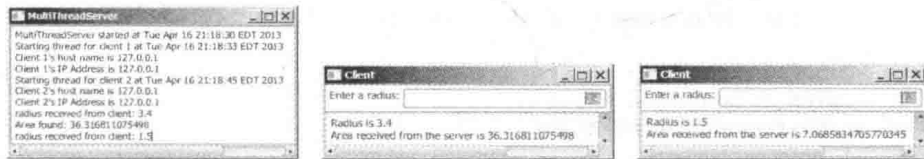


图 31-9 服务器创建一个线程服务一个客户端

程序清单 31-4 MultiThreadServer.java

```

1  import java.io.*;
2  import java.net.*;
3  import java.util.Date;
4  import javafx.application.Application;
5  import javafx.application.Platform;
6  import javafx.scene.Scene;
7  import javafx.scene.control.ScrollPane;
8  import javafx.scene.control.TextArea;
9  import javafx.stage.Stage;
10
11 public class MultiThreadServer extends Application {
12     // Text area for displaying contents
13     private TextArea ta = new TextArea();
14
15     // Number a client
16     private int clientNo = 0;
17
18     @Override // Override the start method in the Application class
19     public void start(Stage primaryStage) {
20         // Create a scene and place it in the stage
21         Scene scene = new Scene(new ScrollPane(ta), 450, 200);
22         primaryStage.setTitle("MultiThreadServer"); // Set the stage title
23         primaryStage.setScene(scene); // Place the scene in the stage
24         primaryStage.show(); // Display the stage
25
26         new Thread(() -> {
27             try {
28                 // Create a server socket
29                 ServerSocket serverSocket = new ServerSocket(8000);
30                 ta.appendText("MultiThreadServer started at "
31                     + new Date() + '\n');
32
33                 while (true) {
34                     // Listen for a new connection request
35                     Socket socket = serverSocket.accept();
36
37                     // Increment clientNo
38                     clientNo++;
39
40                     Platform.runLater(() -> {
41                         // Display the client number
42                         ta.appendText("Starting thread for client " + clientNo +
43                             " at " + new Date() + '\n');
44
45                         // Find the client's host name, and IP address
46                         InetAddress inetAddress = socket.getInetAddress();
47                         ta.appendText("Client " + clientNo + "'s host name is "
48                             + inetAddress.getHostName() + '\n');
49                         ta.appendText("Client " + clientNo + "'s IP Address is "
50                             + inetAddress.getHostAddress() + '\n');
51                     });
52
53                     // Create and start a new thread for the connection

```

```

54         new Thread(new HandleAClient(socket)).start();
55     }
56 }
57 catch(IOException ex) {
58     System.err.println(ex);
59 }
60 }).start();
61 }
62
63 // Define the thread class for handling new connection
64 class HandleAClient implements Runnable {
65     private Socket socket; // A connected socket
66
67     /** Construct a thread */
68     public HandleAClient(Socket socket) {
69         this.socket = socket;
70     }
71
72     /** Run a thread */
73     public void run() {
74         try {
75             // Create data input and output streams
76             DataInputStream inputFromClient = new DataInputStream(
77                 socket.getInputStream());
78             DataOutputStream outputToClient = new DataOutputStream(
79                 socket.getOutputStream());
80
81             // Continuously serve the client
82             while (true) {
83                 // Receive radius from the client
84                 double radius = inputFromClient.readDouble();
85
86                 // Compute area
87                 double area = radius * radius * Math.PI;
88
89                 // Send area back to the client
90                 outputToClient.writeDouble(area);
91
92                 Platform.runLater(() -> {
93                     ta.appendText("radius received from client: " +
94                         radius + '\n');
95                     ta.appendText("Area found: " + area + '\n');
96                 });
97             }
98         }
99         catch(IOException ex) {
100             ex.printStackTrace();
101         }
102     }
103 }
104 }

```

服务器在端口 8000 上创建一个服务器套接字 (第 29 行), 并等待连接 (第 35 行)。当建立一个与客户端的连接后, 服务器就创建一个新线程来处理通信 (第 54 行)。然后, 它在无限的 while 循环中等待另一次连接 (第 33 ~ 55 行)。

互相独立运行的线程与指定的客户端进行通信。每个线程创建数据输入输出流向客户端发送和接收数据。

✓ 复习题

31.8 如何让一个服务器服务多个客户端?

31.5 发送和接收对象

🔑 要点提示：一个程序可以向另一程序发送和接收对象。

在前面的例子中，学习了如何发送和接收基本类型的数据。也可以在套接字流上使用 `ObjectOutputStream` 和 `ObjectInputStream` 来发送和接收对象。为了能够进行传输，这些对象必须是可序列化的。下面的例子将演示如何发送和接收对象。

这个例子包括三个类：`StudentAddress.java`（程序清单 31-5）、`StudentClient.java`（程序清单 31-6）和 `StudentServer.java`（程序清单 31-7）。客户端程序从客户端采集学生信息，并将这些信息发送给服务器，如图 31-10 所示。

`StudentAddress` 类包含学生信息：`name`（姓名）、`street`（街道）、`city`（城市）、`state`（州）和 `zip`（邮编）。

`StudentAddress` 类实现了 `Serializable` 接口。因此，可以使用对象输出流和输入流来发送和接收对象。

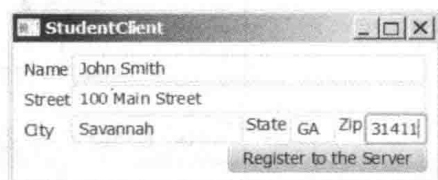


图 31-10 客户端向服务器发送一个对象中的学生信息

程序清单 31-5 `StudentAddress.java`

```
1 public class StudentAddress implements java.io.Serializable {
2     private String name;
3     private String street;
4     private String city;
5     private String state;
6     private String zip;
7
8     public StudentAddress(String name, String street, String city,
9         String state, String zip) {
10         this.name = name;
11         this.street = street;
12         this.city = city;
13         this.state = state;
14         this.zip = zip;
15     }
16
17     public String getName() {
18         return name;
19     }
20
21     public String getStreet() {
22         return street;
23     }
24
25     public String getCity() {
26         return city;
27     }
28
29     public String getState() {
30         return state;
31     }
32
33     public String getZip() {
34         return zip;
35     }
36 }
```

客户端通过输出流套接字上的 `ObjectOutputStream` 发送 `StudentAddress` 对象，服务器

通过输入流套接字上的 `ObjectInputStream` 接收 `Student` 对象, 如图 31-11 所示。客户端使用 `ObjectOutputStream` 类中的 `writeObject` 方法, 向服务器发送学生相关的数据。服务器使用 `ObjectInputStream` 类中的 `readObject` 方法来接收学生信息。服务器程序和客户端程序分别在程序清单 31-6 和程序清单 31-7 中给出。

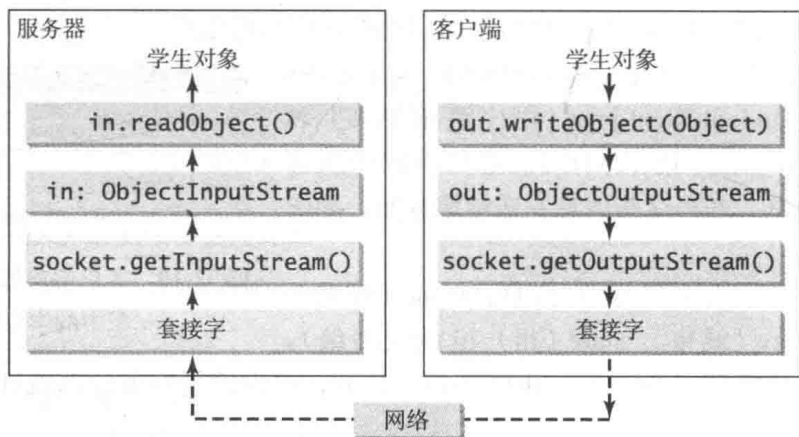


图 31-11 客户端向服务器发送一个 `StudentAddress` 对象

程序清单 31-6 StudentClient.java

```

1  import java.io.*;
2  import java.net.*;
3  import javafx.application.Application;
4  import javafx.event.ActionEvent;
5  import javafx.event.EventHandler;
6  import javafx.geometry.HPos;
7  import javafx.geometry.Pos;
8  import javafx.scene.Scene;
9  import javafx.scene.control.Button;
10 import javafx.scene.control.Label;
11 import javafx.scene.control.TextField;
12 import javafx.scene.layout.GridPane;
13 import javafx.scene.layout.HBox;
14 import javafx.stage.Stage;
15
16 public class StudentClient extends Application {
17     private TextField tfName = new TextField();
18     private TextField tfStreet = new TextField();
19     private TextField tfCity = new TextField();
20     private TextField tfState = new TextField();
21     private TextField tfZip = new TextField();
22
23     // Button for sending a student to the server
24     private Button btRegister = new Button("Register to the Server");
25
26     // Host name or ip
27     String host = "localhost";
28
29     @Override // Override the start method in the Application class
30     public void start(Stage primaryStage) {
31         GridPane pane = new GridPane();
32         pane.add(new Label("Name"), 0, 0);
33         pane.add(tfName, 1, 0);
34         pane.add(new Label("Street"), 0, 1);
35         pane.add(tfStreet, 1, 1);
36         pane.add(new Label("City"), 0, 2);
37
38         // ... (rest of the code for the GUI and network logic)
39     }
40 }
  
```

```

38     HBox hBox = new HBox(2);
39     pane.add(hBox, 1, 2);
40     hBox.getChildren().addAll(tfCity, new Label("State"), tfState,
41         new Label("Zip"), tfZip);
42     pane.add(btRegister, 1, 3);
43     GridPane.setHalignment(btRegister, HPos.RIGHT);
44
45     pane.setAlignment(Pos.CENTER);
46     tfName.setPrefColumnCount(15);
47     tfStreet.setPrefColumnCount(15);
48     tfCity.setPrefColumnCount(10);
49     tfState.setPrefColumnCount(2);
50     tfZip.setPrefColumnCount(3);
51
52     btRegister.setOnAction(new ButtonListener());
53
54     // Create a scene and place it in the stage
55     Scene scene = new Scene(pane, 450, 200);
56     primaryStage.setTitle("StudentClient"); // Set the stage title
57     primaryStage.setScene(scene); // Place the scene in the stage
58     primaryStage.show(); // Display the stage
59 }
60
61 /** Handle button action */
62 private class ButtonListener implements EventHandler<ActionEvent> {
63     @Override
64     public void handle(ActionEvent e) {
65         try {
66             // Establish connection with the server
67             Socket socket = new Socket(host, 8000);
68
69             // Create an output stream to the server
70             ObjectOutputStream toServer =
71                 new ObjectOutputStream(socket.getOutputStream());
72
73             // Get text field
74             String name = tfName.getText().trim();
75             String street = tfStreet.getText().trim();
76             String city = tfCity.getText().trim();
77             String state = tfState.getText().trim();
78             String zip = tfZip.getText().trim();
79
80             // Create a Student object and send to the server
81             StudentAddress s =
82                 new StudentAddress(name, street, city, state, zip);
83             toServer.writeObject(s);
84         }
85         catch (IOException ex) {
86             ex.printStackTrace();
87         }
88     }
89 }
90 }

```

程序清单 31-7 StudentServer.java

```

1  import java.io.*;
2  import java.net.*;
3
4  public class StudentServer {
5      private ObjectOutputStream outputToFile;
6      private ObjectInputStream inputFromClient;
7
8      public static void main(String[] args) {
9          new StudentServer();

```



```
10 }
11
12 public StudentServer() {
13     try {
14         // Create a server socket
15         ServerSocket serverSocket = new ServerSocket(8000);
16         System.out.println("Server started ");
17
18         // Create an object output stream
19         outputToFile = new ObjectOutputStream(
20             new FileOutputStream("student.dat", true));
21
22         while (true) {
23             // Listen for a new connection request
24             Socket socket = serverSocket.accept();
25
26             // Create an input stream from the socket
27             inputFromClient =
28                 new ObjectInputStream(socket.getInputStream());
29
30             // Read from input
31             Object object = inputFromClient.readObject();
32
33             // Write to the file
34             outputToFile.writeObject(object);
35             System.out.println("A new student object is stored");
36         }
37     }
38     catch(ClassNotFoundException ex) {
39         ex.printStackTrace();
40     }
41     catch(IOException ex) {
42         ex.printStackTrace();
43     }
44     finally {
45         try {
46             inputFromClient.close();
47             outputToFile.close();
48         }
49         catch (Exception ex) {
50             ex.printStackTrace();
51         }
52     }
53 }
54 }
```

在客户端，当用户单击 Register to the Server 按钮时，客户端创建一个连接到主机的套接字（第 67 行），在套接字的输出流上创建一个 ObjectOutputStream 对象（第 70 和 71 行），并通过对象输出流调用 writeObject 方法将 StudentAddress 对象发送给服务器（第 83 行）。

在服务器端，当客户端连接到服务器后，服务器在套接字的输入流上创建一个 ObjectInputStream 对象（第 27 和 28 行），通过对象输入流调用 readObject 方法接收 StudentAddress 对象（第 31 行），并把这个对象写到文件中（第 34 行）。

✓ 复习题

- 31.9 服务器怎样接收客户端的连接请求？客户端如何连接到服务器？
- 31.10 如何从服务器端得到客户程序的主机名？
- 31.11 如何发送和接收一个对象？

31.6 示例学习：分布式井字游戏

要点提示：本节开发一个程序，使得两个玩家可以在 Internet 上玩井字游戏。

在 16.12 节中，开发了一个井字游戏的程序，该游戏允许两个游戏者在同一台机器上玩游戏。在本节中，学习如何利用套接字数据流，使用多线程和网络开发一个分布式的井字游戏。分布式井字游戏允许用户在因特网上任意位置的不同机器上玩游戏。

在此需要开发一个多用户服务器。服务器创建一个服务器套接字，并接受每两个玩家一组连接请求，构成一个会话。每个会话都是一个线程，管理两个玩家之间的通信并且判断游戏状态。服务器可以建立任意多个会话，如图 31-12 所示。

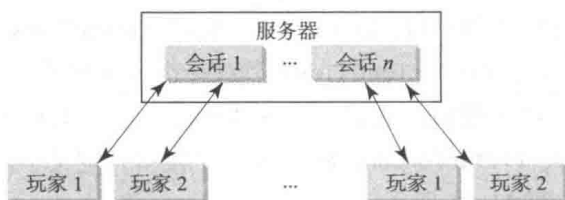


图 31-12 服务器可以创建多个会话，每个会话都为两个玩家提供一局井字游戏

在每一个会话中，第一个与服务器连接的客户端标识为玩家 1，使用的棋子标记为 X，第二个与服务器连接的客户端标识为玩家 2，使用的棋子标记为 O。服务器通知玩家各自使用的标记。一旦两个客户端都与服务器建立连接，服务器就启动一个线程，通过重复执行图 31-13 所示的步骤，实现两个玩家的游戏。

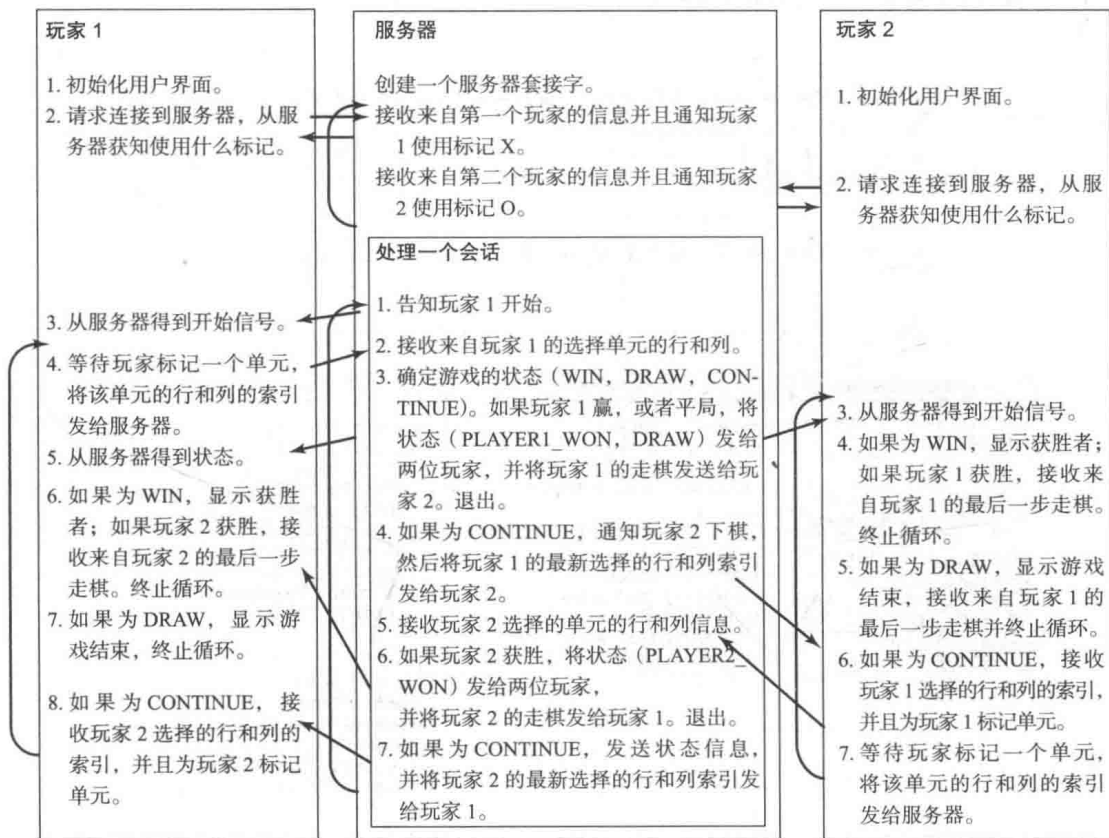


图 31-13 服务器启动一个线程便于两个玩家之间的通信

服务器可以不使用图形组件，但是把它创建成显示游戏信息的 GUI 可以让界面更加友好。可以在 GUI 中创建一个包含文本域的滚动窗格，并在文本域内显示游戏信息。当两个玩家连接到服务器时，服务器就创建一个线程处理游戏会话。

客户端负责与玩家交互。它创建了一个包含 9 个单元的用户界面，并在标签中为用户显示游戏名称和游戏状况。这个客户类与程序清单 16-13 中的 TicTacToe 类非常相似。然而，本例中的客户端并没有判断游戏的状态（输赢或平局），它只是把走棋步骤传给服务器并从服务器接收游戏状态。

基于以上分析，可以创建下面的类：

- 在程序清单 31-9 中，TicTacToeServer 类为所有的客户端提供服务。
- HandleASession 类帮助两个玩家进行游戏。它在 TicTacToeServer.java 中定义。
- 在程序清单 31-10 中，TicTacToeClient 类对一个玩家建模。
- Cell 类对游戏中的单元建模。它是 TicTacToeClient 类的内部类。
- 在程序清单 31-8 中，TicTacToeConstants 是一个接口，定义了该例中所有类共享的常量。这些类之间的关系如图 31-14 所示。

程序清单 31-8 TicTacToeConstants.java

```

1 public interface TicTacToeConstants {
2     public static int PLAYER1 = 1; // Indicate player 1
3     public static int PLAYER2 = 2; // Indicate player 2
4     public static int PLAYER1_WON = 1; // Indicate player 1 won
5     public static int PLAYER2_WON = 2; // Indicate player 2 won
6     public static int DRAW = 3; // Indicate a draw
7     public static int CONTINUE = 4; // Indicate to continue
8 }

```

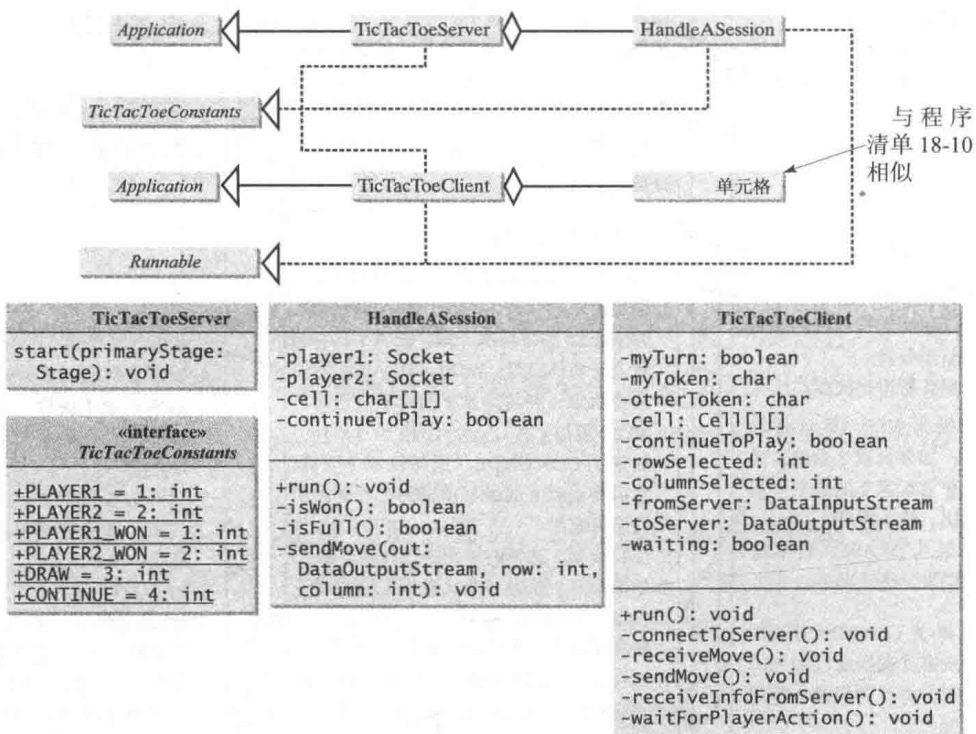


图 31-14 TicTacToeServer 为每一个由两个玩家构成的会话创建一个 HandleASession 实例。TicTacToeClient 在用户界面中创建了 9 个单元

程序清单 31-9 TicTacToeServer.java

```
1  import java.io.*;
2  import java.net.*;
3  import java.util.Date;
4  import javafx.application.Application;
5  import javafx.application.Platform;
6  import javafx.scene.Scene;
7  import javafx.scene.control.ScrollPane;
8  import javafx.scene.control.TextArea;
9  import javafx.stage.Stage;
10
11 public class TicTacToeServer extends Application
12     implements TicTacToeConstants {
13     private int sessionNo = 1; // Number a session
14
15     @Override // Override the start method in the Application class
16     public void start(Stage primaryStage) {
17         TextArea taLog = new TextArea();
18
19         // Create a scene and place it in the stage
20         Scene scene = new Scene(new ScrollPane(taLog), 450, 200);
21         primaryStage.setTitle("TicTacToeServer"); // Set the stage title
22         primaryStage.setScene(scene); // Place the scene in the stage
23         primaryStage.show(); // Display the stage
24
25         new Thread( () -> {
26             try {
27                 // Create a server socket
28                 ServerSocket serverSocket = new ServerSocket(8000);
29
30                 Platform.runLater(() -> taLog.appendText(new Date() +
31                     ": Server started at socket 8000\n"));
32
33                 // Ready to create a session for every two players
34                 while (true) {
35                     Platform.runLater(() -> taLog.appendText(new Date() +
36                         ": Wait for players to join session " + sessionNo + '\n'));
37
38                     // Connect to player 1
39                     Socket player1 = serverSocket.accept();
40
41                     Platform.runLater(() -> {
42                         taLog.appendText(new Date() + ": Player 1 joined session "
43                             + sessionNo + '\n');
44                         taLog.appendText("Player 1's IP address" +
45                             player1.getInetAddress().getHostAddress() + '\n');
46                     });
47
48                     // Notify that the player is Player 1
49                     new DataOutputStream(
50                         player1.getOutputStream()).writeInt(PLAYER1);
51
52                     // Connect to player 2
53                     Socket player2 = serverSocket.accept();
54
55                     Platform.runLater(() -> {
56                         taLog.appendText(new Date() +
57                             ": Player 2 joined session " + sessionNo + '\n');
58                         taLog.appendText("Player 2's IP address" +
59                             player2.getInetAddress().getHostAddress() + '\n');
60                     });
61
62                     // Notify that the player is Player 2
63                     new DataOutputStream(
```

```

63         player2.getOutputStream()).writeInt(PAYER2);
64
65         // Display this session and increment session number
66         Platform.runLater(() ->
67             taLog.appendText(new Date() +
68                 ": Start a thread for session " + sessionNo++ + '\n'));
69
70         // Launch a new thread for this session of two players
71         new Thread(new HandleASession(player1, player2)).start();
72     }
73 }
74 catch(IOException ex) {
75     ex.printStackTrace();
76 }
77 }).start();
78 }
79
80 // Define the thread class for handling a new session for two players
81 class HandleASession implements Runnable, TicTacToeConstants {
82     private Socket player1;
83     private Socket player2;
84
85     // Create and initialize cells
86     private char[][] cell = new char[3][3];
87
88     private DataInputStream fromPlayer1;
89     private DataOutputStream toPlayer1;
90     private DataInputStream fromPlayer2;
91     private DataOutputStream toPlayer2;
92
93     // Continue to play
94     private boolean continueToPlay = true;
95
96     /** Construct a thread */
97     public HandleASession(Socket player1, Socket player2) {
98         this.player1 = player1;
99         this.player2 = player2;
100
101         // Initialize cells
102         for (int i = 0; i < 3; i++)
103             for (int j = 0; j < 3; j++)
104                 cell[i][j] = ' ';
105     }
106
107     /** Implement the run() method for the thread */
108     public void run() {
109         try {
110             // Create data input and output streams
111             DataInputStream fromPlayer1 = new DataInputStream(
112                 player1.getInputStream());
113             DataOutputStream toPlayer1 = new DataOutputStream(
114                 player1.getOutputStream());
115             DataInputStream fromPlayer2 = new DataInputStream(
116                 player2.getInputStream());
117             DataOutputStream toPlayer2 = new DataOutputStream(
118                 player2.getOutputStream());
119
120             // Write anything to notify player 1 to start
121             // This is just to let player 1 know to start
122             toPlayer1.writeInt(1);
123
124             // Continuously serve the players and determine and report
125             // the game status to the players
126             while (true) {
127                 // Receive a move from player 1

```

```
128         int row = fromPlayer1.readInt();
129         int column = fromPlayer1.readInt();
130         cell[row][column] = 'X';
131
132         // Check if Player 1 wins
133         if (isWon('X')) {
134             toPlayer1.writeInt(PAYER1_WON);
135             toPlayer2.writeInt(PAYER1_WON);
136             sendMove(toPlayer2, row, column);
137             break; // Break the loop
138         }
139         else if (isFull()) { // Check if all cells are filled
140             toPlayer1.writeInt(DRAW);
141             toPlayer2.writeInt(DRAW);
142             sendMove(toPlayer2, row, column);
143             break;
144         }
145         else {
146             // Notify player 2 to take the turn
147             toPlayer2.writeInt(CONTINUE);
148
149             // Send player 1's selected row and column to player 2
150             sendMove(toPlayer2, row, column);
151         }
152
153         // Receive a move from Player 2
154         row = fromPlayer2.readInt();
155         column = fromPlayer2.readInt();
156         cell[row][column] = 'O';
157
158         // Check if Player 2 wins
159         if (isWon('O')) {
160             toPlayer1.writeInt(PAYER2_WON);
161             toPlayer2.writeInt(PAYER2_WON);
162             sendMove(toPlayer1, row, column);
163             break;
164         }
165         else {
166             // Notify player 1 to take the turn
167             toPlayer1.writeInt(CONTINUE);
168
169             // Send player 2's selected row and column to player 1
170             sendMove(toPlayer1, row, column);
171         }
172     }
173 }
174 catch(IOException ex) {
175     ex.printStackTrace();
176 }
177 }
178
179 /** Send the move to other player */
180 private void sendMove(DataOutputStream out, int row, int column)
181     throws IOException {
182     out.writeInt(row); // Send row index
183     out.writeInt(column); // Send column index
184 }
185
186 /** Determine if the cells are all occupied */
187 private boolean isFull() {
188     for (int i = 0; i < 3; i++)
189         for (int j = 0; j < 3; j++)
190             if (cell[i][j] == ' ')
191                 return false; // At least one cell is not filled
192 }
```

```

193     // All cells are filled
194     return true;
195 }
196
197 /** Determine if the player with the specified token wins */
198 private boolean isWon(char token) {
199     // Check all rows
200     for (int i = 0; i < 3; i++)
201         if ((cell[i][0] == token)
202             && (cell[i][1] == token)
203             && (cell[i][2] == token)) {
204             return true;
205         }
206
207     /** Check all columns */
208     for (int j = 0; j < 3; j++)
209         if ((cell[0][j] == token)
210             && (cell[1][j] == token)
211             && (cell[2][j] == token)) {
212             return true;
213         }
214
215     /** Check major diagonal */
216     if ((cell[0][0] == token)
217         && (cell[1][1] == token)
218         && (cell[2][2] == token)) {
219         return true;
220     }
221
222     /** Check subdiagonal */
223     if ((cell[0][2] == token)
224         && (cell[1][1] == token)
225         && (cell[2][0] == token)) {
226         return true;
227     }
228
229     /** All checked, but no winner */
230     return false;
231 }
232 }
233 }

```

程序清单 31-10 TicTacToeClient.java

```

1  import java.io.*;
2  import java.net.*;
3  import java.util.Date;
4  import javafx.application.Application;
5  import javafx.application.Platform;
6  import javafx.scene.Scene;
7  import javafx.scene.control.Label;
8  import javafx.scene.control.ScrollPane;
9  import javafx.scene.control.TextArea;
10 import javafx.scene.layout.BorderPane;
11 import javafx.scene.layout.GridPane;
12 import javafx.scene.layout.Pane;
13 import javafx.scene.paint.Color;
14 import javafx.scene.shape.Ellipse;
15 import javafx.scene.shape.Line;
16 import javafx.stage.Stage;
17
18 public class TicTacToeClient extends Application
19     implements TicTacToeConstants {
20     // Indicate whether the player has the turn
21     private boolean myTurn = false;

```



```
22
23 // Indicate the token for the player
24 private char myToken = ' ';
25
26 // Indicate the token for the other player
27 private char otherToken = ' ';
28
29 // Create and initialize cells
30 private Cell[][] cell = new Cell[3][3];
31
32 // Create and initialize a title label
33 private Label lblTitle = new Label();
34
35 // Create and initialize a status label
36 private Label lblStatus = new Label();
37
38 // Indicate selected row and column by the current move
39 private int rowSelected;
40 private int columnSelected;
41
42 // Input and output streams from/to server
43 private DataInputStream fromServer;
44 private DataOutputStream toServer;
45
46 // Continue to play?
47 private boolean continueToPlay = true;
48
49 // Wait for the player to mark a cell
50 private boolean waiting = true;
51
52 // Host name or ip
53 private String host = "localhost";
54
55 @Override // Override the start method in the Application class
56 public void start(Stage primaryStage) {
57     // Pane to hold cell
58     GridPane pane = new GridPane();
59     for (int i = 0; i < 3; i++)
60         for (int j = 0; j < 3; j++)
61             pane.add(cell[i][j] = new Cell(i, j), j, i);
62
63     BorderPane borderPane = new BorderPane();
64     borderPane.setTop(lblTitle);
65     borderPane.setCenter(pane);
66     borderPane.setBottom(lblStatus);
67
68     // Create a scene and place it in the stage
69     Scene scene = new Scene(borderPane, 320, 350);
70     primaryStage.setTitle("TicTacToeClient"); // Set the stage title
71     primaryStage.setScene(scene); // Place the scene in the stage
72     primaryStage.show(); // Display the stage
73
74     // Connect to the server
75     connectToServer();
76 }
77
78 private void connectToServer() {
79     try {
80         // Create a socket to connect to the server
81         Socket socket = new Socket(host, 8000);
82
83         // Create an input stream to receive data from the server
84         fromServer = new DataInputStream(socket.getInputStream());
85
86         // Create an output stream to send data to the server
```

```

87     toServer = new DataOutputStream(socket.getOutputStream());
88 }
89 catch (Exception ex) {
90     ex.printStackTrace();
91 }
92
93 // Control the game on a separate thread
94 new Thread(() -> {
95     try {
96         // Get notification from the server
97         int player = fromServer.readInt();
98
99         // Am I player 1 or 2?
100        if (player == PLAYER1) {
101            myToken = 'X';
102            otherToken = 'O';
103            Platform.runLater(() -> {
104                lblTitle.setText("Player 1 with token 'X'");
105                lblStatus.setText("Waiting for player 2 to join");
106            });
107
108            // Receive startup notification from the server
109            fromServer.readInt(); // Whatever read is ignored
110
111            // The other player has joined
112            Platform.runLater(() -> {
113                lblStatus.setText("Player 2 has joined. I start first");
114            });
115
116            // It is my turn
117            myTurn = true;
118        }
119        else if (player == PLAYER2) {
120            myToken = 'O';
121            otherToken = 'X';
122            Platform.runLater(() -> {
123                lblTitle.setText("Player 2 with token 'O'");
124                lblStatus.setText("Waiting for player 1 to move");
125            });
126        }
127
128        // Continue to play
129        while (continueToPlay) {
130            if (player == PLAYER1) {
131                waitForPlayerAction(); // Wait for player 1 to move
132                sendMove(); // Send the move to the server
133                receiveInfoFromServer(); // Receive info from the server
134            }
135            else if (player == PLAYER2) {
136                receiveInfoFromServer(); // Receive info from the server
137                waitForPlayerAction(); // Wait for player 2 to move
138                sendMove(); // Send player 2's move to the server
139            }
140        }
141    } catch (Exception ex) {
142        ex.printStackTrace();
143    }
144 }).start();
145 }
146
147 /** Wait for the player to mark a cell */
148 private void waitForPlayerAction() throws InterruptedException {
149     while (waiting) {
150         Thread.sleep(100);
151     }

```

```
152
153     waiting = true;
154 }
155
156 /** Send this player's move to the server */
157 private void sendMove() throws IOException {
158     toServer.writeInt(rowSelected); // Send the selected row
159     toServer.writeInt(columnSelected); // Send the selected column
160 }
161
162 /** Receive info from the server */
163 private void receiveInfoFromServer() throws IOException {
164     // Receive game status
165     int status = fromServer.readInt();
166
167     if (status == PLAYER1_WON) {
168         // Player 1 won, stop playing
169         continueToPlay = false;
170         if (myToken == 'X') {
171             Platform.runLater(() -> lblStatus.setText("I won! (X)"));
172         }
173         else if (myToken == 'O') {
174             Platform.runLater(() ->
175                 lblStatus.setText("Player 1 (X) has won!"));
176             receiveMove();
177         }
178     }
179     else if (status == PLAYER2_WON) {
180         // Player 2 won, stop playing
181         continueToPlay = false;
182         if (myToken == 'O') {
183             Platform.runLater(() -> lblStatus.setText("I won! (O)"));
184         }
185         else if (myToken == 'X') {
186             Platform.runLater(() ->
187                 lblStatus.setText("Player 2 (O) has won!"));
188             receiveMove();
189         }
190     }
191     else if (status == DRAW) {
192         // No winner, game is over
193         continueToPlay = false;
194         Platform.runLater(() ->
195             lblStatus.setText("Game is over, no winner!"));
196
197         if (myToken == 'O') {
198             receiveMove();
199         }
200     }
201     else {
202         receiveMove();
203         Platform.runLater(() -> lblStatus.setText("My turn"));
204         myTurn = true; // It is my turn
205     }
206 }
207
208 private void receiveMove() throws IOException {
209     // Get the other player's move
210     int row = fromServer.readInt();
211     int column = fromServer.readInt();
212     Platform.runLater(() -> cell[row][column].setToken(otherToken));
213 }
214
215 // An inner class for a cell
216 public class Cell extends Pane {
```

```

217 // Indicate the row and column of this cell in the board
218 private int row;
219 private int column;
220
221 // Token used for this cell
222 private char token = ' ';
223
224 public Cell(int row, int column) {
225     this.row = row;
226     this.column = column;
227     this.setPrefSize(2000, 2000); // What happens without this?
228     setStyle("-fx-border-color: black"); // Set cell's border
229     this.setOnMouseClicked(e -> handleMouseClicked());
230 }
231
232 /** Return token */
233 public char getToken() {
234     return token;
235 }
236
237 /** Set a new token */
238 public void setToken(char c) {
239     token = c;
240     repaint();
241 }
242
243 protected void repaint() {
244     if (token == 'X') {
245         Line line1 = new Line(10, 10,
246             this.getWidth() - 10, this.getHeight() - 10);
247         line1.endXProperty().bind(this.widthProperty().subtract(10));
248         line1.endYProperty().bind(this.heightProperty().subtract(10));
249         Line line2 = new Line(10, this.getHeight() - 10,
250             this.getWidth() - 10, 10);
251         line2.startYProperty().bind(
252             this.heightProperty().subtract(10));
253         line2.endXProperty().bind(this.widthProperty().subtract(10));
254
255         // Add the lines to the pane
256         this.getChildren().addAll(line1, line2);
257     }
258     else if (token == 'O') {
259         Ellipse ellipse = new Ellipse(this.getWidth() / 2,
260             this.getHeight() / 2, this.getWidth() / 2 - 10,
261             this.getHeight() / 2 - 10);
262         ellipse.centerXProperty().bind(
263             this.widthProperty().divide(2));
264         ellipse.centerYProperty().bind(
265             this.heightProperty().divide(2));
266         ellipse.radiusXProperty().bind(
267             this.widthProperty().divide(2).subtract(10));
268         ellipse.radiusYProperty().bind(
269             this.heightProperty().divide(2).subtract(10));
270         ellipse.setStroke(Color.BLACK);
271         ellipse.setFill(Color.WHITE);
272
273         getChildren().add(ellipse); // Add the ellipse to the pane
274     }
275 }
276
277 /** Handle a mouse click event */
278 private void handleMouseClicked() {
279     // If cell is not occupied and the player has the turn
280     if (token != ' ' && myTurn) {

```

```

281         setToken(myToken); // Set the player's token in the cell
282         myTurn = false;
283         rowSelected = row;
284         columnSelected = column;
285         lblStatus.setText("Waiting for the other player to move");
286         waiting = false; // Just completed a successful move
287     }
288 }
289 }
290 }

```

服务器可以同时为任意多个会话提供服务。每个会话对应两个玩家。客户端可以部署和运行在 Java applet。要想在 Web 浏览器上运行 Java applet 客户端程序，服务器程序必须在 Web 服务器上运行。图 31-15 和图 31-16 分别显示服务器和客户端的运行示例。

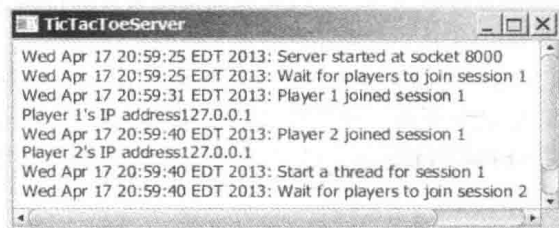


图 31-15 TicTacToeServer 接收连接请求并创建会话来为一对玩家提供服务

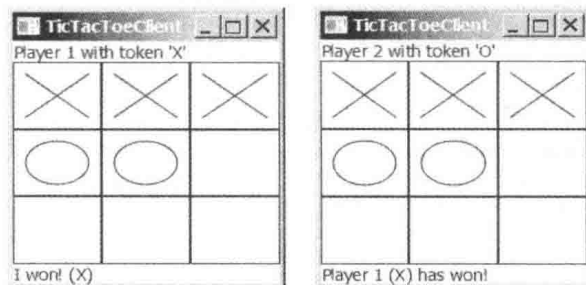


图 31-16 TicTacToeClient 可以作为 applet 或者独立应用运行

TicTacToeConstants 接口定义了程序中所有类共享的常量。每个使用这些常量的类需要实现这个接口。在接口中集中定义常量是 Java 中常用的做法。

会话一旦建立起来，服务器便交替地从玩家那里接收下棋信息。玩家接收下棋信息后，服务器判断游戏的状态。如果游戏没有结束，那么服务器把状态（CONTINUE）和一个玩家的下棋信息发送给另一个玩家。如果游戏是获胜或平局，服务器把状态（PLAYER1_WON、PLAYER2_WON 或 DRAW）发送给两个玩家。

套接字层要实现的 Java 网络程序是严格同步的。从一台机器发送数据的操作要求对应一个从另一台机器接收数据的操作。如本例所示，服务器和客户端都是严格同步发送或接收数据的。

✓ 复习题

- 33.12 如果程序清单 31-10 中第 227 行没有设置单元的优先尺寸，会发生什么？
- 33.13 如果没有轮到一个玩家下棋，但是他点到了一个空的单元上，程序清单 31-10 的客户程序会做什么？

关键术语

Client socket (客户端套接字)	packet-based communication (基于包的通信)
Domain name (域名)	server socket (服务器套接字)
Domain name server (域名服务器)	socket (套接字)
localhost (本地主机)	stream-based communication (基于流的通信)
IP address (IP 地址)	TCP (传输控制协议)
port (端口)	UDP (用户数据报协议)

本章小结

- 1. Java 支持流套接字和数据报套接字。流套接字使用 TCP (传输控制协议) 来进行数据传输。而数据报套接字使用 UDP (用户数据报协议)。由于 TCP 协议检测丢失的传输并重新提交它们, 所以, 传输是无损的和可靠的。相反地, UDP 不能保证传输是无损的。
- 2. 要创建一个服务器, 必须首先使用语句 `new ServerSocket(port)` 获取一个服务器套接字。在创建服务器套接字之后, 可以启动服务器, 使用服务器套接字上的 `accept()` 方法监听连接请求。客户端通过使用 `new socket(serverName,port)` 来创建一个客户端套接字, 用于向服务器发送连接请求。
- 3. 当服务器与客户端的连接建立之后, 流套接字通信与输入输出流通信非常相似。可以通过套接字上的 `getInputStream()` 方法获得一个输入流, 通过 `getOutputStream()` 方法获得一个输出流。
- 4. 一个服务器经常同时与多个客户端协同工作。通过为每个连接创建一个线程, 可以利用线程同时处理服务器的多个客户端。

测试题

回答位于网址 www.cs.armstrong.edu/liang/intro10e/quiz.html 的本章测试题。

编程练习题

31.2 节

*31.1 (贷款服务器) 为一个客户端编写一个服务器。客户端向服务器发送贷款信息 (年利率、贷款年限和贷款总额) (如图 31-17a 所示)。服务器计算月偿还额和总偿还额, 并把它们发回给客户端 (如图 31-17b 所示)。将客户端程序命名为 `Exercise31_01Client`, 将服务器程序命名为 `Exercise31_01Server`。

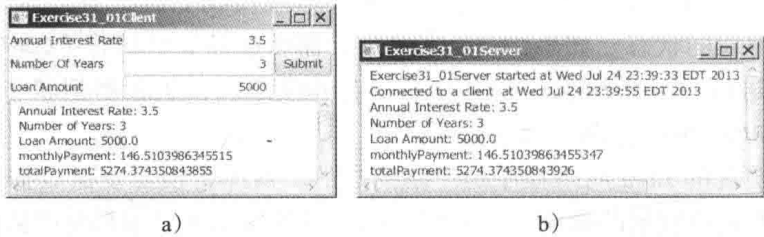


图 31-17 a) 客户端向服务器发送年利率、贷款年限和贷款总额; b) 从服务器接收月偿还额和总偿还额

*31.2 (BMI 服务器) 为一个客户端编写一个服务器。客户端向服务器发送体重和身高信息 (如图 31-18a 所示)。服务器计算 BMI (Body Mass Index, 身体质量指数), 发回一个报告 BMI 的字符串给客户端 (如图 31-18b 所示)。如何计算 BMI 参见 3.8 节。将客户端程序命名为

Exercise31_02Client, 将服务器程序命名为 Exercise31_02Server。

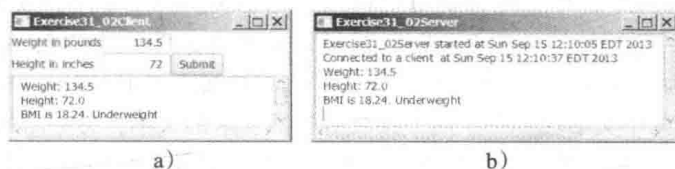


图 31-18 a) 客户端向服务器发送一个人的体重和身高; b) 从服务器接收 BMI

31.3 节和 31.4 节

*31.3 (可用于多客户端的贷款服务器) 修改编程练习题 31.1, 编写一个可以用于多客户端的贷款服务器。

31.5 节

31.4 (对客户计数) 编写一个服务器程序, 追踪连接到服务器的客户数量。当一个新的连接建立的时候, 计数加 1。计数采用随机访问文件存储。编写一个客户程序, 接收从服务器发来的数字并且显示一条消息, 比如 “You are visitor number 11”, 如图 31-19 所示。将客户端程序命名为 Exercise31_04Client, 将服务器程序命名为 Exercise31_04Server。

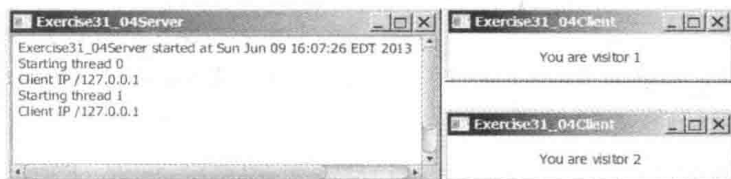


图 31-19 客户端显示服务器访问了多少次, 服务器端存储该计数

31.5 (将贷款信息以对象发送) 修改编程练习题 31.1, 使得客户端可以发送一个贷款对象, 该对象包含了年利率、年数以及贷款数等信息, 对于服务器来说, 可以发送月付和总支付金额。

31.6 节

31.6 (显示和添加地址) 开发一个客户端 / 服务器应用程序, 可以查看和添加地址, 如图 31-20 所示。

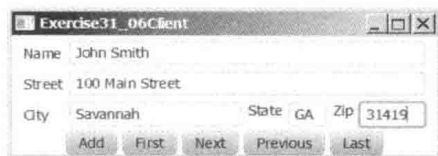


图 31-20 可以查看和添加地址

- 使用代码清单 31-5 中定义的 StudentAddress 类, 在其对象中保存 name (姓名)、street (街道)、city (城市)、state (州) 和 zip (邮编) 等属性。
- 用户可以使用按钮 First、Next、Previous 和 Last 来查看地址, 并且使用按钮 Add 添加新地址。
- 限制同时连接两个客户端。

将客户端程序命名为 Exercise31_06Client, 将服务器程序命名为 Exercise31_06Server。

*31.7 (在数组中传递最后 100 个数字) 编程练习题 22.12 从文件 PrimeNumbers.dat 中获取最后 100 个素数。编写一个客户端程序, 要求服务器发送数组中的最后 100 个素数。将服务器程序命名为 Exercise31_07Server, 将客户端程序命名为 Exercise31_07Client。假设 PrimeNumbers.dat 中的 long 类型的数字以二进制形式存储。

*31.8 (在 ArrayList 中传递最后 100 个数字) 编程练习题 24.12 从名为 PrimeNumbers.dat 的文件中获取最后 100 个素数。编写一个客户端程序, 要求服务器发送 ArrayList 中的最后 100 个素数。将服务器程序命名为 Exercise31_08Server, 将客户端程序命名为 Exercise31_08Client。假设

PrimeNumbers.dat 中的 long 类型的数字以二进制形式存储。

****31.9 (聊天程序)** 编写一个程序，使两个用户可以互相聊天。把一个用户实现为服务器（见图 31-21a），另一个用户实现为客户端（见图 31-21b）。服务器有两个文本域：一个用于输入文本，另一个（不可编辑的）显示从客户端接收的文本。当用户按下 Enter 键时，当前行就被发送给客户端。客户端也有两个文本域：一个用于接收服务器发来的文本，另一个用于输入文本。当用户按下 Enter 键时，当前行被发送给服务器。将客户端程序命名为 Exercise31_09Client，将服务器程序命名为 Exercise31_09Server。



图 31-21 服务器与客户端互相发送或接收文本信息

*****31.10 (多客户端聊天程序)** 编写一个程序，允许任意数目的客户端互相聊天。实现一个服务器为所有的客户端服务，如图 31-22 所示。将客户端程序命名为 Exercise31_10Client，将服务器程序命名为 Exercise31_10Server。

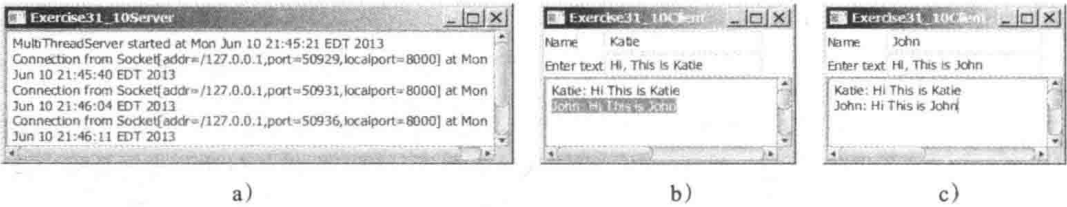


图 30-22 服务器在 a 中启动，它有三个客户端，如 b 和 c 中所示

Java 数据库程序设计

教学目标

- 理解数据库和数据库管理系统的概念（32.2 节）。
- 解关系数据模型：关系数据结构、约束和语言（32.2 节）。
- 使用 SQL 创建和删除表，以及获取和修改数据（32.3 节）。
- 学会使用 JDBC 加载驱动程序、连接数据库、执行语句和处理结果集（32.4 节）。
- 使用预备语句执行预编译的 SQL 语句（32.5 节）。
- 使用可调用语句执行存储的 SQL 过程和函数（32.6 节）。
- 使用 DatabaseMetaData 和 ResultSetMetaData 接口检索数据库元数据（32.7 节）。

32.1 引言

要点提示：Java 提供了开发数据库程序的 API，可以工作于任何的关系型数据库系统之上。

你可能已经了解了许多关于数据库系统的知识。数据库系统无处不在。例如，个人的社会保障信息存储在政府的数据库中；如果你在网上购物，你的购物信息就存储在网上商店的数据库中；如果你上大学，你的学籍信息就存储在学校数据库中。数据库系统不仅存储数据，还提供访问、更新、处理和分析数据的方法。例如，社会保障信息周期性地更新，可以在网上注册课程。数据库系统在社会和商业中起着重要的作用。

本章将介绍数据库系统、SQL 以及如何使用 Java 开发数据库应用程序。如果你已经了解 SQL，就可以跳过 32.2 节和 32.3 节。

32.2 关系型数据库系统

要点提示：SQL 是定义和访问数据库的标准数据库语言。

数据库系统（database system）由数据库、存储和管理数据库中数据的软件，以及显示数据并让用户能够与数据库系统进行交互的应用程序组成，如图 32-1 所示。

数据库是由构成信息的数据组成的存储。当你从软件发行商那里购买一个数据库系统，例如，MySQL、Oracle、IBM 的 DB2 以及 Informix、Microsoft SQL Server 或 Sybase 等，实际上是购买了一个构成数据库管理系统（database management system, DBMS）的软件。数据库管理系统是为专业程序设计人员的使用而设计的，并不适合普通用户。为了能够使用户访问和更新数据库，需要在 DBMS 之上建立应用程序。因此，可以把应用程序视为数据库系统和用户之间的接口。应用

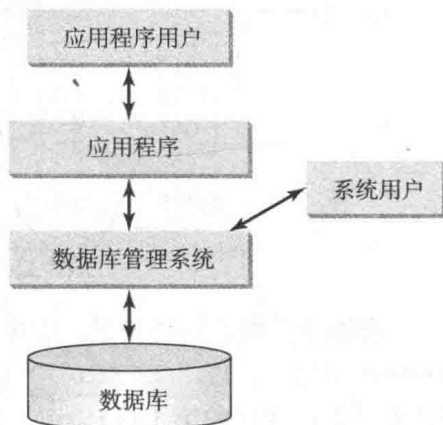


图 32-1 数据库系统由数据、数据库管理软件和应用程序构成

程序可以是单机上的 GUI 应用程序或者 Web 应用程序，并且可以在网络上访问多个不同的数据库系统，如图 32-2 所示。

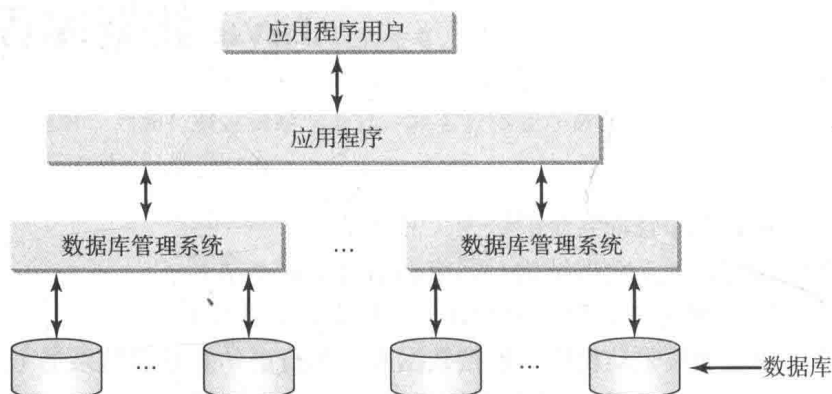


图 32-2 一个应用程序可以访问多个数据库系统

目前，大多数数据库系统都是关系数据库系统（relational database system）。它们都是基于关系数据模型的，这种模型有三个要素：结构、完整性和语言。结构（structure）定义了数据的表示，完整性（integrity）给出一些对数据的约束，语言（language）提供了访问和操纵数据的手段。

32.2.1 关系结构

关系模型是围绕着一个简单自然的结构建立的。一个关系实际上是一个没有重复行的表格。表格很容易理解，也很容易使用。关系模型提供了一种简单但强有力的数据表示方法。

表的一行表示一条记录，表的一列表示该记录中一个属性的值。在关系数据库理论中，一行称为一个元组（tuple），一列称为一个属性（attribute）。图 32-3 显示了一个由某大学提供的存储课程信息的示例表格。该表格有 8 个记录，每个记录有 5 个属性。

关系 / 表格名字		列 / 属性				
课程表格		courseId	subjectId	courseNumber	title	numOfCredits
元组 / 行		11111	CSCI	1301	Introduction to Java I	4
		11112	CSCI	1302	Introduction to Java II	3
		11113	CSCI	3720	Database Systems	3
		11114	CSCI	4750	Rapid Java Application	3
		11115	MATH	2750	Calculus I	5
		11116	MATH	3750	Calculus II	5
		11117	EDUC	1111	Reading	3
		11118	ITEC	1344	Database Administration	3

图 32-3 一张表格具有表名、列名和行

表描述数据之间的关系。表中的每一行表示相互关联的数据构成的一条记录。例如，表 Course 中的“11111”“CSCI”“1301”“Introduction to Java I”和“4”相互关联，构成一条记录（图 32-3 中的第 1 行）。正如同一行中的数据相互关联一样，不同表格中的数据通过共同属性也可能相互关联。假设数据库中有另外两个名为 Student（学生）和 Enrollment（注册）的表，如图 32-4 和图 32-5 所示。表 Course 和表 Enrollment 通过它们共同的属性 courseId

(课程编号) 建立关联，而表 Enrollment 和表 Student 通过 ssn (社保号) 建立关联。

Student Table										
ssn	firstName	mi	lastName	phone	birthDate	street	zipCode	deptID		
444111110	Jacob	R	Smith	9129219434	1985-04-09	99 Kingston Street	31435	BIOL		
444111111	John	K	Stevenson	9129219434	null	100 Main Street	31411	BIOL		
444111112	George	K	Smith	9129213454	1974-10-10	1200 Abercorn St.	31419	CS		
444111113	Frank	E	Jones	9125919434	1970-09-09	100 Main Street	31411	BIOL		
444111114	Jean	K	Smith	9129219434	1970-02-09	100 Main Street	31411	CHEM		
444111115	Josh	R	Woo	7075989434	1970-02-09	555 Franklin St.	31411	CHEM		
444111116	Josh	R	Smith	9129219434	1973-02-09	100 Main Street	31411	BIOL		
444111117	Joy	P	Kennedy	9129229434	1974-03-19	103 Bay Street	31412	CS		
444111118	Toni	R	Peterson	9129229434	1964-04-29	103 Bay Street	31412	MATH		
444111119	Patrick	R	Stoneman	9129229434	1969-04-29	101 Washington St.	31435	MATH		
444111120	Rick	R	Carter	9125919434	1986-04-09	19 West Ford St.	31411	BIOL		

图 32-4 表 Student 存储学生的信息

Enrollment Table			
ssn	courseId	dateRegistered	grade
444111110	11111	2004-03-19	A
444111110	11112	2004-03-19	B
444111110	11113	2004-03-19	C
444111111	11111	2004-03-19	D
444111111	11112	2004-03-19	F
444111111	11113	2004-03-19	A
444111112	11114	2004-03-19	B
444111112	11115	2004-03-19	C
444111112	11116	2004-03-19	D
444111113	11111	2004-03-19	A
444111113	11113	2004-03-19	A
444111114	11115	2004-03-19	B
444111115	11115	2004-03-19	F
444111115	11116	2004-03-19	F
444111116	11111	2004-03-19	D
444111117	11111	2004-03-19	D
444111118	11111	2004-03-19	A
444111118	11112	2004-03-19	D
444111118	11113	2004-03-19	B

图 32-5 表 Enrollment 存储学生的注册信息

32.2.2 完整性约束

完整性约束 (integrity constraint) 对表格强加了一个条件，表中的所有合法值都必须满足该条件。图 32-6 显示了表 Subject 和表 Course 中的一些完整性约束的例子。

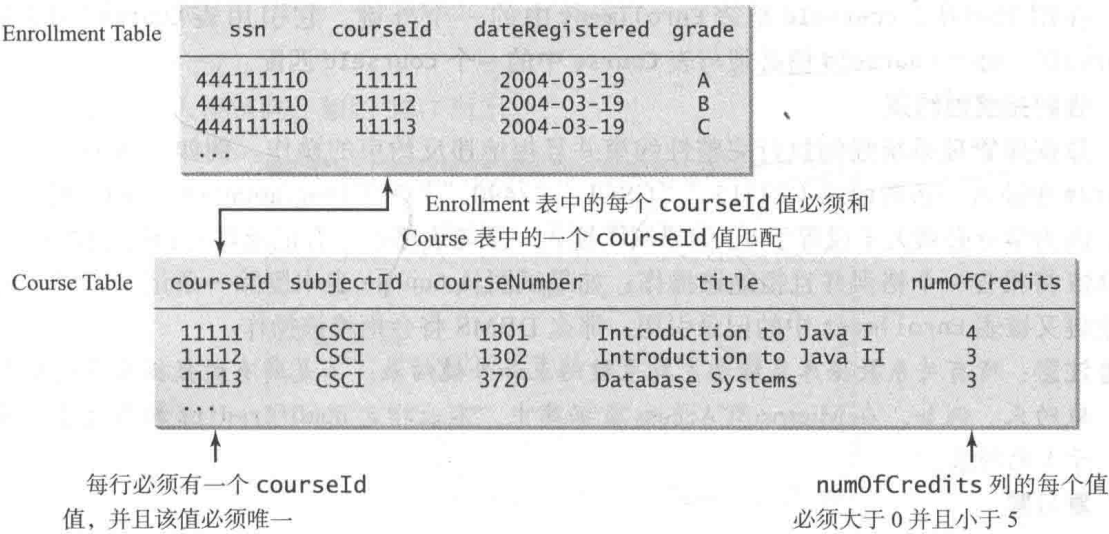


图 32-6 表 Enrollment 和表 Course 具有完整性约束

一般来说,有三种类型的约束:域约束、主键约束和外键约束。域约束(domain constraint)和主键约束(primary key constraint)被称为内部关联型约束(intrarelational constraint),意味着每个约束只涉及一个关系。外键约束(foreign key constraint)是相互关联型的(interrelational),意味着一个涉及多个关系的约束。

域约束

域约束(domain constraint)规定一个属性的允许值。域可以使用标准数据类型来指定,例如,整数、浮点数、定长字符串和变长字符串等。标准数据类型指定的值范围较大,可以指定附加的约束来缩小这个范围。例如,可以指定(Course 表中的)numOfCredits 属性的值必须大于 0 且小于 5,也可以指定一个属性的值能否为空值(null),空值是数据库中的特殊值,表示未知或不可用。例如,表 Student 中的 birthDate 属性的值可以为 null。

主键约束

要理解主键,先了解一下超键、键和候选键的概念是有帮助的。超键(superkey)是一个属性或一组属性,它唯一地标识了一个关系。也就是说,没有两个记录具有相同的超键值。由定义可知,一个关系是由一组互相不同的记录组成的。关系中的所有属性的集合构成一个超键。

键(key) K 是一个最小的超键,意思是指 K 的任何真子集都不是超键。一个关系可以有几个键,在这种情况下,每个键都称为一个候选键(candidate key)。主键(primary key)是由数据库设计者指定的候选键之一,通常用来标识一个关系中的记录。在图 32-6 中, courseId 是 Course 表中的主键。

外键约束

在关系数据库中,数据是相互关联的。关系中的记录是相互关联的,而不同关系中的记录通过它们的共同属性也是相互关联的。简单地说,共同属性就是外键。外键约束(foreign key constraint)定义了关系之间的关系。


形式化的话,如果属性集 FK 满足下面两条规则,则 FK 是关系 R 的一个外键(foreign key),它引用关系 T :

- FK 中的属性与关系 T 中的主键具有相同的域。
- 关系 R 中 FK 的非空值必须与关系 T 中的一个主键值相匹配。

在图 32-6 中, courseId 是表 Enrollment 中的一个外键,它引用表 Course 中的主键 courseId。每个 courseId 值必须与表 Course 中的一个 courseId 匹配。

强制完整性约束

数据库管理系统强制执行完整性约束并且拒绝违反约束的操作。例如,如果试图向表 Course 中插入一条新记录(“1115,” “CSCI,” “2490,” “C++ Programming,” 0),则不会成功,因为学分必须大于或等于 0;如果试图插入一条与表格中已有记录具有相同主键的记录,DBMS 将报告一个错误并且拒绝该操作;如果试图从 Course 表中删除一条记录,而该记录的主键又被表 Enrollment 中的记录引用,那么 DBMS 将会拒绝该操作。

 **注意:** 所有关系数据库系统都支持主键约束和外键约束。不是所有的数据库系统都支持域约束。例如,在 Microsoft Access 数据库中,不能指定 numOfCredits 的值大于 0 且小于 5 的约束。

复习题

32.1 什么是超键、候选键和主键?

32.2 什么是外键?

32.3 一个关系可以有多个主键或多个外键吗?

32.4 在同一关系中, 外键必须是主键吗?

32.5 一个外键需要与它的引用主键具有相同的名字吗?

32.6 外键的值可以是空值吗?

32.3 SQL

要点提示: 结构化查询语言 (structured query language, SQL) 是用来定义表格和完整性约束以及访问和操纵数据的语言。

SQL (读作 S-Q-L 或 sequel) 是访问关系数据库系统的通用语言。应用程序也许允许用户不直接使用 SQL 也可以访问数据库, 但是, 这些应用程序本身一定是使用 SQL 访问数据库的。本节将介绍一些基本的 SQL 命令。

注意: 关系数据库管理系统有很多种。它们共享相同的 SQL 语言, 但是不一定支持 SQL 的每个特征。一些系统对 SQL 语言进行了扩展。本节介绍所有系统都支持的标准 SQL 语言。

SQL 可以用于 MySQL、Oracle、Sybase、IBM DB2、IBM Informix、MS Access 或者任何其他关系数据库系统。本章使用 MySQL 来演示 SQL, 并且使用 MySQL、Access 和 Oracle 来演示 Java 数据库程序设计。本书的 Web 网站包含了关于如何安装和使用 MySQL、Oracle 和 Access 这三种流行数据库系统的内容:

- 补充材料 IV.B: MySQL 教程。
- 补充材料 IV.C: Oracle 教程。
- 补充材料 IV.D: Microsoft Access 教程。

32.3.1 在 MySQL 上创建用户账户

假定你已经按照默认配置安装好了 MySQL 5, 为了匹配本书中所有的例子, 应该创建一个账户, 账户名为 scott, 密码为 tiger。可以使用 MySQL Workbench 管理工具或命令行执行管理任务。MySQL Workbench 是管理 MySQL 数据库的 GUI 工具。下面是从命令行创建账户的步骤:

1) 在 DOS 命令行提示符下, 输入

```
mysql -uroot -p
```

系统提示输入根密码, 如图 32-7 所示。

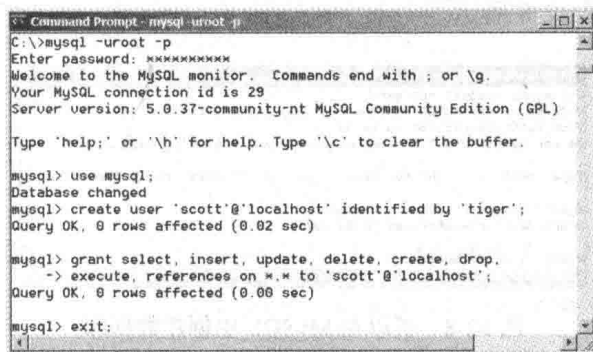


图 32-7 你可以通过命令窗口访问 MySQL 数据库服务器

2) 在 mysql 提示符下, 输入

```
use mysql;
```

3) 要创建用户名为 scott, 密码为 tiger 的账户, 输入

```
create user 'scott'@'localhost' identified by 'tiger';
```

4) 赋予特权给 scott, 键入

```
grant select, insert, update, delete, create, create view, drop,  
execute, references on *.* to 'scott'@'localhost';
```

- 如果希望该账号可以从任意的 IP 地址来远程访问, 输入

```
grant all privileges on *.* to 'scott'@'%'  
identified by 'tiger';
```

- 如果希望限制该账号从一个特定的 IP 地址可以进行远程访问, 输入

```
grant all privileges on *.* to 'scott'@'ipAddress'  
identified by 'tiger';
```

5) 输入

```
exit;
```

退出 MySQL 控制台。

注意: 在 Windows 系统中, 每次计算机启动时自动启动 MySQL 数据库服务器。输入命令 `net stop mysql` 可以终止它, 输入命令 `net start mysql` 能够重新启动它。

默认情况下, 该服务器包含两个名为 `mysql` 和 `test` 的数据库。`mysql` 数据库包含存储服务器信息及其用户信息的表格, 它是为服务器管理员的使用而设计的。例如, 管理员可以使用它创建用户、授予和撤销用户权限。既然你是系统中所装服务器的主人, 你就拥有了对 `mysql` 数据库的全部访问权限, 但是不能在 `mysql` 数据库中创建用户表。可以使用 `test` 数据库来存储数据或者创建新的数据库, 也可以使用命令 `create database databasename` 创建一个新的数据库或者用命令 `drop database databasename` 删除一个已经存在的数据库。

32.3.2 创建数据库

为了匹配本书中的例子, 需要创建一个名为 `javabook` 的数据库。下面是创建它的步骤:

1) 在 DOS 命令行提示符下, 输入

```
mysql -uscott -ptiger
```

登录到 mysql, 如图 32-8 所示。

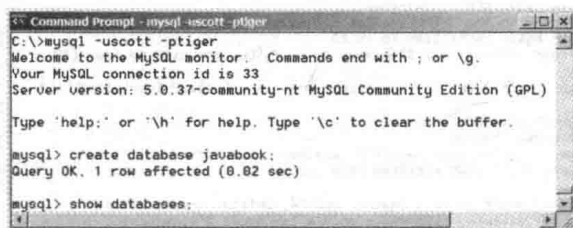


图 32-8 可以在 MySQL 中创建数据库

2) 在 mysql 提示符下, 输入


```
create database javabook;
```

为了方便起见, 补充材料 IV.A 提供了本书中创建和初始化表格所用的 SQL 语句。你可以下载这些 MySQL 脚本并将其存入 **script.sql**。为了执行这个脚本文件, 首先要用命令下面的命令转到 **javabook** 数据库:

```
use javabook;
```

然后输入

```
source script.sql;
```

如图 32-9 所示。

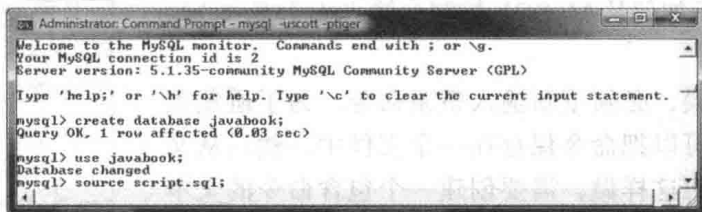


图 32-9 可以在脚本文件中运行 SQL 命令

注意: 可以使用补充材料 IV.A 中的脚本来填充 **javabook** 数据库。

32.3.3 创建和删除表

表是数据库中最基本的对象。要创建一个表, 可以使用 **create table** 语句指定表名、属性以及类型, 如下例所示:

```
create table Course (
    courseId char(5),
    subjectId char(4) not null,
    courseNumber integer,
    title varchar(50) not null,
    numOfCredits integer,
    primary key (courseId)
);
```

这条语句创建了一个名为 **Course** 的表, 它包含属性 **courseId**、**subjectId**、**courseNumber**、**title** 和 **numOfCredits**。每个属性都有一个数据类型, 该数据类型规定了属性中存储的数据的类型。**char(5)** 表明 **courseId** 由 5 个字符组成, **varchar(50)** 表明 **title** 是一个字符个数最多为 50 的变长字符串, **integer** 表明 **courseNumber** 是一个整数。主键是 **courseId**。

表 **Student** 和表 **Enrollment** 可以按如下方式创建:

```
create table Student (
    ssn char(9),
    firstName varchar(25),
    mi char(1),
    lastName varchar(25),
    birthDate date,
    street varchar(25),
    phone char(11),
    zipCode char(5),
    deptId char(4),
    primary key (ssn)
);

create table Enrollment (
    ssn char(9),
    courseId char(5),
    dateRegistered date,
    grade char(1),
    primary key (ssn, courseId),
    foreign key (ssn) references
        Student(ssn),
    foreign key (courseId) references
        Course(courseId)
);
```

注意：SQL 的关键字不区分大小写。本书采用下面的命名规则：表格的命名方式与 Java 类的命名方式一样，属性的命名方式和 Java 变量的命名方式一样。SQL 关键字的命名方式和 Java 关键字的命名方式相同。

如果不再需要一个表，可以使用 `drop table` 命令把它永久地删除。例如，可以使用下面的语句删除表 `Course`：

```
drop table Course;
```

如果要删除的表被其他表引用，必须先删除其他表。例如，如果已经创建了表 `Course`、`Student` 和 `Enrollment`，要删除表 `Course`，必须先删除表 `Enrollment`，因为表 `Enrollment` 引用了表 `Course`。

图 32-10 演示了如何从 MySQL 控制台输入 `create table` 语句。

如果输入有错误，必须重新输入整条命令。为了避免重新输入整条命令，可以把命令保存在一个文件中，然后从文件中执行命令。要想这样做，需要创建一个包含命令的文本文件，例如，名为 `test.sql` 的文件。可以使用像 Notepad 这样的文本编辑器来创建文本文件，如图 32-11a 所示。要为某行命令添加注释，可在注释的前面加上两个破折号。现在，可以在 SQL 命令提示符下输入 `source test.sql` 来运行这个脚本文件，如图 32-11b 所示。

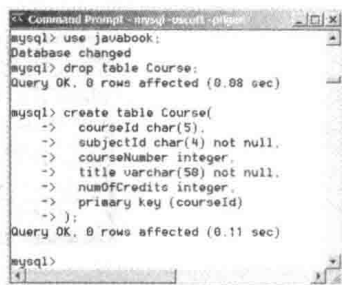


图 32-10 使用 `create table` 语句创建一个表

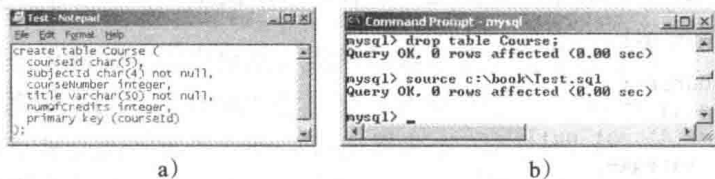


图 32-11 a) 可以使用记事本创建保存 SQL 命令的文本文件；b) 可以在 MySQL 中运行脚本文件内的 SQL 命令

32.3.4 简单插入、更新和删除

一旦创建了表格，就能够向它插入数据，也可以更新或删除记录。本节介绍简单的插入、更新和删除语句。

在表中插入一条记录的语法是：

```
insert into tableName [(column1, column2, ..., column)]
values (value1, value2, ..., valuen);
```

例如，下面的语句在表 `Course` 中插入一条记录。新记录的 `courseId` 是 '11113'，`subjectId` 是 'CSCI'，`courseNumber` 是 '3720'，`title` 是 'Database Systems'，`creditHours` 是 '3'。

```
insert into Course (courseId, subjectId, courseNumber, title, numOfCredits)
values ('11113', 'CSCI', '3720', 'Database Systems', 3);
```

列名是可选的。尽管列具有默认值，但是如果省略了列名，必须输入记录中所有列的值。在 SQL 语言中，字符串的值是区分大小写的，并且包含在单引号中。

更新表格的语法是：

```
update tableName
set column1 = newValue1 [, column2 = newValue2, ...]
[where condition];
```

例如，下面的语句把 title 为 Database Systems 的课程的 numOfCredits 值改为 4：

```
update Course
set numOfCredits = 4
where title = 'Database Systems';
```

从表中删除记录的语法是：

```
delete from tableName
[where condition];
```

例如，下面的语句从表 Course 中删除课程 Database Systems：

```
delete from Course
where title = 'Database Systems';
```

下面的语句删除表 Course 中的所有记录：

```
delete from Course;
```

32.3.5 简单查询

要从表中获取信息，所使用的 select 语句需要遵循下面的语法：

```
select column-list
from table-list
[where condition];
```

select 子句列出所选定的列。from 子句指定查询所涉及的表。可选的 where 子句指明选择行的条件。

查询 1：查找 CS 系（计算机科学系）的所有学生，查询结果如图 32-12 所示。

```
select firstName, mi, lastName
from Student
where deptId = 'CS';
```

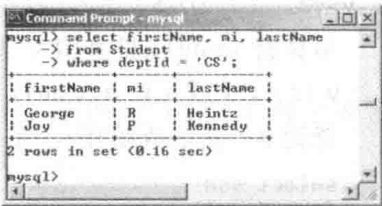


图 32-12 select 语句的执行结果显示在 MySQL 控制台中

32.3.6 比较运算符和布尔运算符

SQL 有 6 个比较运算符（如表 32-1 所示）和 3 个布尔运算符（如表 32-2 所示）。

表 32-1 比较运算符

运算符	描述	运算符	描述	运算符	描述
=	等于	<	小于	>	大于
<> or !=	不等于	<=	小于或等于	>=	大于或等于

表 32-2 布尔运算符

运算符	描述	运算符	描述	运算符	描述
not	逻辑非	and	逻辑与	or	逻辑或

注意：SQL 中的比较运算符和布尔运算符与 Java 中的意义相同。在 SQL 中相等运算符是 =，但在 Java 中是 ==。在 SQL 中不等于运算符是 <> 或 !=，但在 Java 中是 !=。逻辑

非、逻辑与和逻辑或运算符在 Java 中是 !、&& (&) 和 || (|)。

查询 2: 查找 CS 系并且居住区邮编 (zipCode) 为 31411 的学生名单:

```
select firstName, mi, lastName
from Student
where deptId = 'CS' and zipCode = '31411';
```

{ } 注意: 要选择表中的所有属性, 不需要在 select 子句中列出所有属性的名字, 而只需要使用一个星号 (*), 用它表示所有的属性, 例如, 下面的查询显示 CS 系且居住区邮编为 31411 的学生的所有属性。

```
select *
from Student
where deptId = 'CS' and zipCode = '31411';
```

32.3.7 操作符 like、between-and 和 is null

SQL 有一个可以用于模式匹配的操作符 like。检验字符串 s 是否含有模式 p 的语法是:

s like p 或 s not like p

在模式 p 中可以使用通配符 % (百分号) 和 _ (下划线)。% 匹配零个或多个字符, _ 与 s 中的任何单字符匹配。例如, lastName like '_mi%' 表示与第二个和第三个字符分别为 m 和 i 在任意字符串匹配。lastName not like '_mi%' 表示排除第二个和第三个字符分别是 m 和 i 的任意字符串。

{ } 注意: 在 MS Access 的早期版本中, 通配符是 *, 而字符 ? 与任意单字符匹配。

运算符 between-and 检查值 v 是否在值 v1 和 v2 之间, 使用如下语法:

v between v1 and v2 或 v not between v1 and v2

v between v1 and v2 等价于 v >= v1 and v <= v2, v not between v1 and v2 等价于 v < v1 or v > v2。

运算符 is null 检查值 v 是否为 null (空), 使用如下语法:

v is null 或 v is not null

查询 3: 获取成绩在 C 到 A 之间的学生的社会保险号码 ssn:

```
select ssn
from Enrollment
where grade between 'C' and 'A';
```

32.3.8 列的别名

当显示查询结果时, SQL 使用列名作为列的标题。通常, 用户对列采用缩写名, 而且当创建表时列名没有空格。有时希望在结果标题中给出更具描述性的名字, 可以通过下面的语法使用列的别名:

```
select columnName [as] alias
```

查询 4: 获取 CS 系学生的姓氏 (last name) 和邮编。将列名 lastName 显示为 Last Name, 将列名 zipCode 显示为 Zip Code。查询结果如图 32-13 所示。

```
select lastName as "Last Name", zipCode as "Zip Code"
from Student
where deptId = 'CS';
```

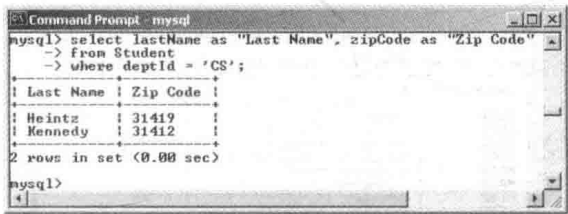


图 32-13 在显示时可以采用别名

{ } 注意：关键字 as 在 MySQL 和 Oracle 中是可选的，但是在 MS Access 中是必需的。

32.3.9 算术运算符

在 SQL 中可以使用算术运算符 * (乘法)、/ (除法)、+ (加法) 和 - (减法)。

查询 5：假设一个学分代表 50 分钟的授课，求出 CSCI 学科每门课程的总分钟数。查询结果如图 32-14 所示。

```
select title, 50 * numOfCredits as "Lecture Minutes Per Week"
from Course
where subjectId = 'CSCI';
```

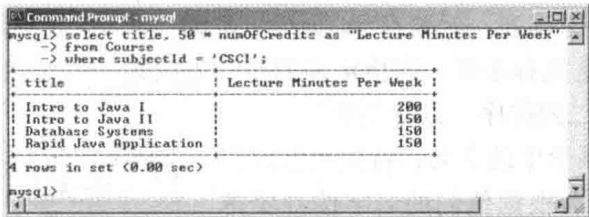


图 32-14 可以在 SQL 中使用算术运算符

32.3.10 显示互不相同的记录

SQL 提供关键字 distinct，可以用于去除输出重复的元组。例如，图 32-15a 显示课程使用的所有课程 ID，图 32-15b 显示课程使用的所有唯一的课程 ID。

```
select distinct subjectId as "Subject ID"
from Course;
```

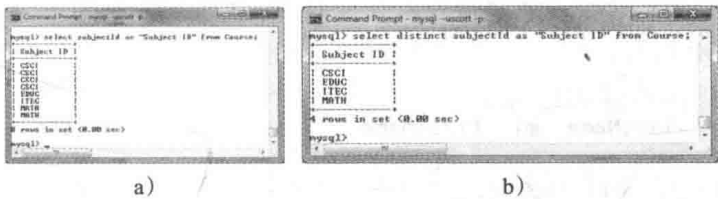


图 32-15 a) 显示重复的记录；b) 显示不同的记录

当 select 子句中条目多于一项时，关键字 distinct 可以查找所有条目的相异记录。例如，下面的语句显示所有具有不同 subjectId 和 title 的记录，如图 32-16 所示。注意，有些记录可能具有相同的 subjectId，但是不同的 title。这些记录是不同的。

```
select distinct subjectId, title
from Course;
```

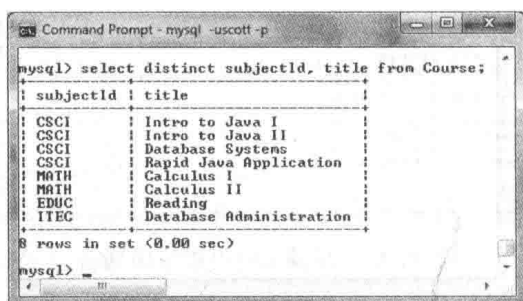


图 32-16 关键字 distinct 应用于整条记录

32.3.11 显示排好序的记录

SQL 提供对输出结果排序的 order by 子句，语法如下：

```
select column-list
from table-list
[where condition]
[order by columns-to-be-sorted];
```

在这个语法结构中，columns-to-be-sorted 指定要排序的一列或多列。默认情况下，是按升序排列的。要按降序排列，就要在 columns-to-be-sorted 之后附加关键字 desc。也可以追加关键字 asc，但是没有必要。当指定多列时，先按第一列对各行排序，然后对第一列具有相同值的行再按第二列排序，以此类推。

查询 6：列出 CS 系学生的全名，首先根据他们的姓氏按降序排列，然后根据他们的名字按升序排列。查询结果如图 32-17 所示。

```
select lastName, firstName, deptId
from Student
where deptId = 'CS'
order by lastName desc, firstName asc;
```

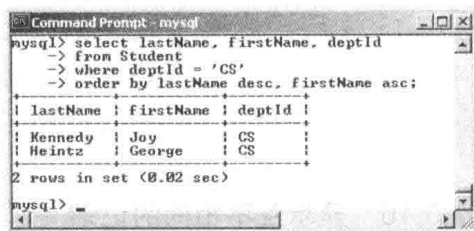


图 32-17 使用 order by 子句对结果排序

32.3.12 联结表

经常会需要从多个表中获取信息，如下面的查询所示。

查询 7：列出学生 Jacob Smith 所学的课程。要完成这个查询，需要将表 Student 和表 Enrollment 联结起来，如图 32-18 所示。

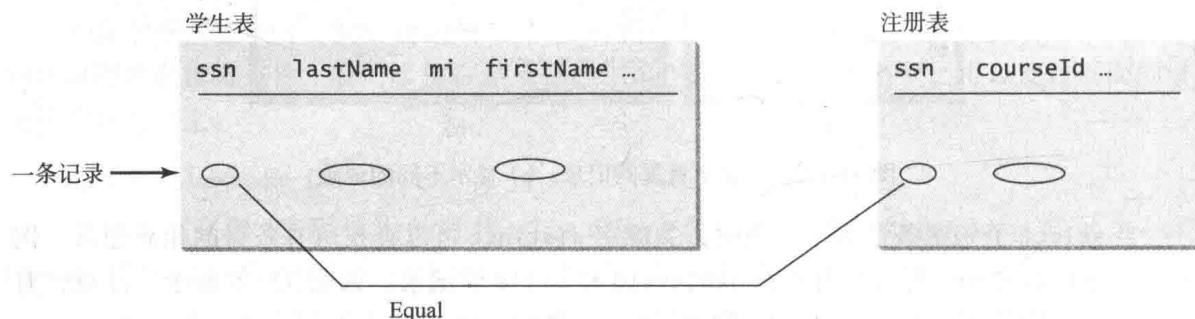
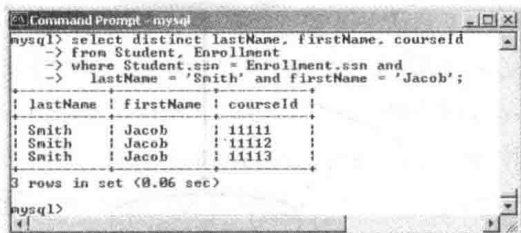


图 32-18 表 Student 和表 Enrollment 通过属性 ssn 联结

可以用 SQL 语言编写下面的查询：

```
select distinct lastName, firstName, courseId
from Student, Enrollment
where Student.ssn = Enrollment.ssn and
      lastName = 'Smith' and firstName = 'Jacob';
```

from 子句中列出了表 Student 和表 Enrollment。这个查询检查每一行对，这一行对是由表 Student 中的一个条目与表 Enrollment 中的另一个条目组成的，该查询选出满足 where 子句所列条件的行对。表 Student 中的行具有姓 Smith 和名 Jacob，表 Student 和表 Enrollment 中的行具有相同的 ssn 值。对选中的每一行对，来自表 Student 的 lastName 和 firstName，与来自表 Enrollment 的 courseId 用来产生结果，如图 32-19 所示。表 Student 和表 Enrollment 具有相同的属性 ssn。要在一个查询中区分它们，可以使用 Student.ssn 和 Enrollment.ssn。



lastName	firstName	courseId
Smith	Jacob	11111
Smith	Jacob	11112
Smith	Jacob	11113

图 32-19 查询 7 演示涉及多个表的查询

对于 SQL 的更多特性，参见补充材料 IV.H 和 IV.I。

复习题

- 32.7 使用 32.3.3 节中的 create table 语句，创建表 Course、Student 和 Enrollment。使用图 32-3 ~ 图 32-5 中的数据在表 Course、Student 和 Enrollment 中插入行。
- 32.8 列出至少含有 4 个学分的所有 CSCI 课程。
- 32.9 列出姓氏中含有两个字母 e 的所有学生。
- 32.10 列出生日为空的所有学生。
- 32.11 列出选修数学 (Math) 课程的所有学生。
- 32.12 列出每个学科的课程数目。
- 32.13 假设每个学分是 50 分钟的授课，求出每个学生所学课程的总分钟数。

32.4 JDBC

要点提示：JDBC 是访问关系型数据库的 Java API。

开发数据库应用程序的 Java API 称为 JDBC。JDBC 是一种 Java API 的商标名称，它支持访问关系数据库的 Java 程序。JDBC 不是首字母的缩写词，但是常被认为表示 Java 数据库连接 (Java database connectivity)。

JDBC 给 Java 程序员提供访问和操纵众多关系数据库的一个统一接口。使用 JDBC API，用 Java 程序设计语言编写的应用程序能够以一种用户友好的接口执行 SQL 语句、获取结果以及显示数据，并且可以将所做的改动传回数据库。JDBC API 还可用于与分布式、异构环境中的多种数据源之间实现交互。

图 32-20 显示了 Java 程序、JDBC API、JDBC 驱动程序和关系数据库之间的关系。JDBC API 是一个 Java 接口和类的集合，用于编写访问和操纵关系数据库的 Java 程序。JDBC 驱动程序起着接口的作用，它使 JDBC 与具体数据库之间的通信灵活方便。它是与具体数据库相关的并且通常由数据库厂商提供。访问 MySQL 数据库需要使用 MySQL JDBC 驱动程序，而访问 Oracle 数据库需要使用 Oracle JDBC 驱动程序。对于 Access 数据库，需要使用包含在 JDK 中的 JDBC-ODBC 桥式驱动程序。ODBC 是 Microsoft 开发的一种技术，用于访问 Windows 平台的数据库。Windows 中预装了 ODBC 驱动程序。JDBC-ODBC 桥式驱动程序使 Java 程序可以访问任何 ODBC 数据源。

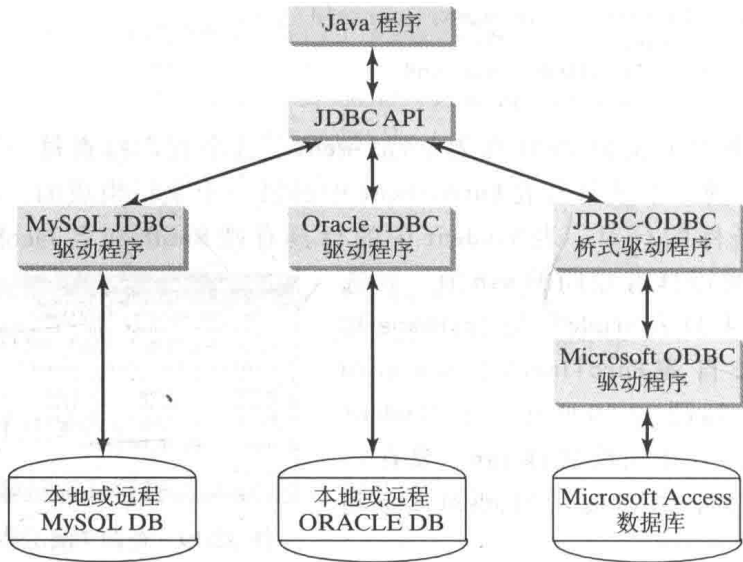


图 32-20 Java 程序通过 JDBC 驱动程序访问和操纵数据库

32.4.1 使用 JDBC 开发数据库应用程序

JDBC API 是一个针对通用 SQL 数据库的 Java 应用编程接口，使 Java 开发者能够使用一致的接口开发独立于 DBMS 的 Java 应用程序。

JDBC API 由类和接口构成，这些类和接口用于建立数据库的连接、把 SQL 语句发送到数据库、处理 SQL 语句的结果以及获取数据库的元数据。使用 Java 开发任何数据库应用程序都需要 4 个主要接口：Driver、Connection、Statement 和 ResultSet。这些接口定义了使用 SQL 访问数据库的一般架构。JDBC API 定义了这些接口。JDBC 驱动程序开发商为这些接口提供实现。程序员使用这些接口。

这些接口的关系如图 32-21 所示。JDBC 应用程序使用 Driver 接口加载一个合适的驱动程序，使用 Connection 接口连接到数据库，使用 Statement 接口创建和执行 SQL 语句，如果语句返回结果，那么使用 ResultSet 接口处理结果。注意，有一些语句不返回结果，例如，SQL 数据定义语句和 SQL 数据修改语句。

JDBC 接口和类是开发 Java 数据库程序的构建模块。访问数据库的典型 Java 程序主要采用下列步骤：

- 1) 加载驱动程序。
在连接到数据库之前，必须使用下面的语句，加载一个合适的驱动程序。

```
Class.forName("JDBCdriverClass");
```

驱动程序是一个实现接口 java.sql.Driver 的具体类。表 32-3 列出了 Access、MySQL 和 Oracle 的驱动程序。如果程序访问一些不同的数据库，必须加载它们各自的驱动程序。

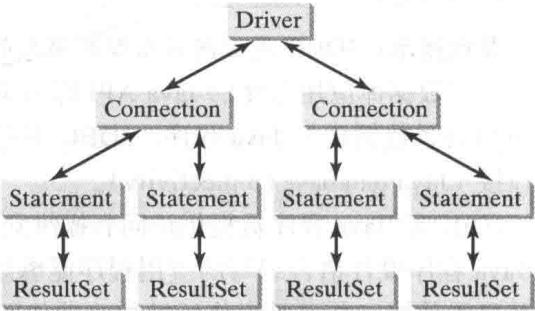


图 32-21 JDBC 类使得 Java 程序连接到数据库，发送 SQL 语句以及处理结果

表 32-3 JDBC 驱动程序

数据库	驱动程序类	来源
Access	sun.jdbc.odbc.JdbcOdbcDriver	已经在 JDK 中
MySQL	com.mysql.jdbc.Driver	mysql-connector-java-5.1.26.jar
Oracle	oracle.jdbc.driver.OracleDriver	ojdbc6.jar

Access 的 JDBC-ODBC 驱动程序捆绑在 JDK 中。当前最新的平台独立的 MySQL JDBC 驱动程序是 mysql-connector-java-5.1.26.jar。该文件包含在可从网址 dev.mysql.com/downloads/connector/j/ 下载的 ZIP 文件中。目前最新的 Oracle JDBC 驱动程序为 ojdbc6.jar (可从网址 www.oracle.com/technetwork/database/enterprise-edition/jdbc-112010-090769.html 下载)。为了使用 MySQL 和 Oracle 驱动程序，必须将 mysql-connector-java-5.1.26.jar 和 ojdbc6.jar 添加到类路径中，在 Windows 中使用下面的 DOS 命令进行添加：

```
set classpath=%classpath%;c:\book\lib\mysql-connector-java-5.1.26.jar;  
c:\book\lib\ojdbc6.jar
```

如果使用 IDE，比如 Eclipse 或者 NetBean，需要添加这些 jar 文件到 IDE 的库中。

[] 注意：com.mysql.jdbc.Driver 是 mysql-connector-java-5.1.26.jar 中的一个类，oracle.jdbc.driver.OracleDriver 是 ojdbc6.jar 中的一个类。mysql-connector-java-5.1.26.jar 和 ojdbc6.jar 包含许多支持驱动程序的类。这些类由 JDBC 使用，但不直接由 JDBC 程序员使用。当你在程序中明确使用某个类时，它被 JVM 自动加载。但是在程序中不显式地使用驱动程序类，因此，必须编写代码告诉 JVM 加载它们。

[] 注意：Java 6 支持驱动程序的自动加载，因此不需要显式地加载它们。但是，在编写本书的时候，并不是所有的驱动程序都有这个特性。为安全起见，应该显式加载驱动程序。

2) 建立连接。

为了连接到一个数据库，需要使用 DriverManager 类中的静态方法 getConnection(databaseURL)，如下所示：

```
Connection connection = DriverManager.getConnection(databaseURL);
```

其中 databaseURL 是数据库在 Internet 上的唯一标识符。表 32-4 列出了数据库 MySQL、Oracle 和 Access 的 URL 模式。

表 32-4 JDBC URLs

数据库	URL 模式
Access	jdbc:odbc:dataSource
MySQL	jdbc:mysql://hostname/dbname
Oracle	jdbc:oracle:thin:@hostname:port#:oracleDBSID

对 ODBC 数据源来说，databaseURL 是 jdbc:odbc:dataSource。ODBC 数据源可以使用 Windows 下的 ODBC 数据源管理器来创建。关于如何创建 Access 数据库的 ODBC 数据源，参见补充材料 IV.D。

假如已经为一个 Access 数据库创建了一个名为 ExampleMDBDataSource 的数据源，使用下面的语句创建一个 Connection 对象：

```
Connection connection = DriverManager.getConnection  
("jdbc:odbc:ExampleMDBDataSource");
```

MySQL 数据库的 `databaseURL` 指定定位数据库的主机名和数据库名。例如，下面的语句以用户名 `scott` 和密码 `tiger`，为本地 MySQL 数据库 `javabook` 创建一个 `Connection` 对象：

```
Connection connection = DriverManager.getConnection
("jdbc:mysql://localhost/javabook", "scott", "tiger");
```

回顾一下，MySQL 默认包含两个名为 `mysql` 和 `test` 的数据库。在 32.3.2 节中，我们创建了一个名为 `javabook` 的自定义数据库，现在在这个例子中使用该数据库。

Oracle 数据库的 `databaseURL` 指定主机名 (`hostname`)、数据库监听输入连接请求的端口号 (`port#`)，以及定位数据库的数据库名 (`oracleDBSID`)。例如，下面的语句为 Oracle 数据库创建一个 `Connection` 对象，主机为 `liang.armstrong.edu`，用户名为 `scott`，口令为 `tiger`：

```
Connection connection = DriverManager.getConnection
("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl",
"scott", "tiger");
```

3) 创建语句。

如果把一个 `Connection` 对象想象成一条连接程序和数据库的缆道，那么 `Statement` 的对象可以看做一辆缆车，它为数据库传输 SQL 语句用于执行，并把运行结果返回程序。一旦创建了 `Connection` 对象，就可以创建执行 SQL 语句的语句，如下所示：

```
Statement statement = connection.createStatement();
```

4) 执行语句。

可以使用方法 `executeUpdate(String sql)` 来执行 SQL DDL (数据定义语言) 或更新语句，可以使用 `executeQuery(String sql)` 来执行 SQL 查询语句。查询结果在 `ResultSet` 中返回。例如，下面的代码执行 SQL 语句 `create table Temp(col1 char(5), col2 char(5))`：

```
statement.executeUpdate
("create table Temp (col1 char(5), col2 char(5))");
```

下面的代码执行 SQL 查询 `select firstName, mi, lastName from Student where lastName = 'Smith'`：

```
// Select the columns from the Student table
ResultSet resultSet = statement.executeQuery
("select firstName, mi, lastName from Student where lastName "
+ " = 'Smith'");
```

5) 处理 `ResultSet`。

结果集 `ResultSet` 维护一个表，该表的当前行可以获得。当前行的初始位置是 `null`。可以使用 `next` 方法移动到下一行，可以使用各种 `getter` 方法从当前行获取值。例如，下面给出的代码显示前面 SQL 查询的所有结果。

```
// Iterate through the result and print the student names
while (resultSet.next())

System.out.println(resultSet.getString(1) + " " +
resultSet.getString(2) + " " + resultSet.getString(3));
```

方法 `getString(1)`、`getString(2)` 和 `getString(3)` 分别获取 `firstName` 列、`mi` 列和 `lastName` 列的值。还可使用 `getString("firstName")`、`getString("mi")` 和 `getString("lastName")` 来获取同样的三列值。第一次执行 `next()` 方法时，将当前行设置为结果集中的第一行，接着再调用 `next()` 方法，将当前行设为第二行，然后是第三行，以此类推，直到最后一行。

程序清单 32-1 是一个使用 JDBC 的例子，完整地演示了连接数据库、执行简单查询以及处理查询结果的过程。该程序连接到一个本地 MySQL 数据库，然后显示姓为 Smith 的全部学生。

程序清单 32-1 SimpleJDBC.java

```
1 import java.sql.*;
2
3 public class SimpleJdbc {
4     public static void main(String[] args)
5         throws SQLException, ClassNotFoundException {
6         // Load the JDBC driver
7         Class.forName("com.mysql.jdbc.Driver");
8         System.out.println("Driver loaded");
9
10        // Connect to a database
11        Connection connection = DriverManager.getConnection
12            ("jdbc:mysql://localhost/javabook", "scott", "tiger");
13        System.out.println("Database connected");
14
15        // Create a statement
16        Statement statement = connection.createStatement();
17
18        // Execute a statement
19        ResultSet resultSet = statement.executeQuery
20            ("select firstName, mi, lastName from Student where LastName "
21            + " = 'Smith'");
22
23        // Iterate through the result and print the student names
24        while (resultSet.next())
25            System.out.println(resultSet.getString(1) + "\t" +
26                resultSet.getString(2) + "\t" + resultSet.getString(3));
27
28        // Close the connection
29        connection.close();
30    }
31 }
```

第 7 行的语句为 MySQL 加载一个 JDBC 驱动程序，第 11 ~ 13 行的语句连接到一个本地 MySQL 数据库，也可以修改它们使其连接到 Access 数据库或 Oracle 数据库。程序创建一个 Statement 对象（第 16 行），执行 SQL 语句并返回一个 ResultSet 对象（第 19 ~ 21 行），然后从 ResultSet 对象获得查询结果（第 24 ~ 26 行）。最后一条语句（第 29 行）关闭连接并释放与连接有关的资源。可以使用 try-with-resources 语法重写该程序。参见 www.cs.armstrong.edu/liang/intro10e/html/SimpleJdbcWithAutoClose.html。

注意：如果在 DOS 提示符下运行本程序，请在 classpath 中指定合适的驱动程序，如图 32-22 所示。

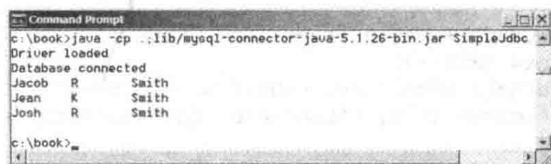


图 32-22 必须包含驱动程序文件以运行 Java 数据库程序

classpath 目录和 jar 文件用逗号分隔。句点 (.) 表示当前目录。为了方便起见，驱动器文件放在 c:\book\lib 目录下。

{ } 警告：在 Java 程序中，不要使用分号（；）结束 Oracle SQL 命令。分号不能用于 Oracle JDBC 驱动程序，但是它可以用于本书中使用的其他驱动程序。

{ } 注意：Connection 接口处理事务并指定它们是如何处理的。默认情况下，新连接是自动提交模式，并且每条 SQL 语句都作为一个单独的事务执行和提交。提交发生在一条语句完成或下次执行发生时，不管哪一个先发生。对于返回结果集的语句，语句完成是指已经获取了结果集的最后一行或结果集已经关闭。如果一条语句返回多个结果，那么提交发生在获取所有结果的时候。可以用 `setAutoCommit(false)` 方法取消自动提交，此时，调用方法 `commit()` 或 `rollback()` 之前的所有语句都被组织成一个事务。`rollback()` 方法取消事务引起的所有变化。

32.4.2 从 JavaFX 访问数据库

本节给出一个示例，演示从一个 JavaFX 程序连接数据库。该程序让用户输入 SSN 和课程 ID 来找到学生的成绩，如图 32-23 所示。程序清单 32-2 中的代码使用本地机器上的 MySQL 数据库。

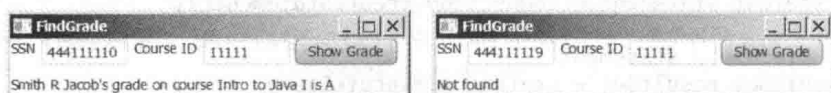


图 32-23 一个 JavaFX 客户端可以访问服务器上的数据库

程序清单 32-2 FindGrade.java

```

1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.scene.control.Label;
5  import javafx.scene.control.TextField;
6  import javafx.scene.layout.HBox;
7  import javafx.scene.layout.VBox;
8  import javafx.stage.Stage;
9  import java.sql.*;
10
11  public class FindGrade extends Application {
12      // Statement for executing queries
13      private Statement stmt;
14      private TextField tfSSN = new TextField();
15      private TextField tfCourseId = new TextField();
16      private Label lblStatus = new Label();
17
18      @Override // Override the start method in the Application class
19      public void start(Stage primaryStage) {
20          // Initialize database connection and create a Statement object
21          initializedDB();
22
23          Button btShowGrade = new Button("Show Grade");
24          HBox hBox = new HBox(5);
25          hBox.getChildren().addAll(new Label("SSN"), tfSSN,
26                                  new Label("Course ID"), tfCourseId, (btShowGrade));
27
28          VBox vBox = new VBox(10);
29          vBox.getChildren().addAll(hBox, lblStatus);
30
31          tfSSN.setPrefColumnCount(6);
32          tfCourseId.setPrefColumnCount(6);
33          btShowGrade.setOnAction(e -> showGrade());
34      }

```

```

35     // Create a scene and place it in the stage
36     Scene scene = new Scene(vBox, 420, 80);
37     primaryStage.setTitle("FindGrade"); // Set the stage title
38     primaryStage.setScene(scene); // Place the scene in the stage
39     primaryStage.show(); // Display the stage
40 }
41
42 private void initializeDB() {
43     try {
44         // Load the JDBC driver
45         Class.forName("com.mysql.jdbc.Driver");
46         // Class.forName("oracle.jdbc.driver.OracleDriver");
47         System.out.println("Driver loaded");
48
49         // Establish a connection
50         Connection connection = DriverManager.getConnection
51             ("jdbc:mysql://localhost/javabook", "scott", "tiger");
52         // ("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl",
53         // "scott", "tiger");
54         System.out.println("Database connected");
55
56         // Create a statement
57         stmt = connection.createStatement();
58     }
59     catch (Exception ex) {
60         ex.printStackTrace();
61     }
62 }
63
64 private void showGrade() {
65     String ssn = tfSSN.getText();
66     String courseId = tfCourseId.getText();
67     try {
68         String queryString = "select firstName, mi, " +
69             "lastName, title, grade from Student, Enrollment, Course " +
70             "where Student.ssn = '" + ssn + "' and Enrollment.courseId = "
71             + "'" + courseId +
72             "' and Enrollment.courseId = Course.courseId " +
73             " and Enrollment.ssn = Student.ssn";
74
75         ResultSet rset = stmt.executeQuery(queryString);
76
77         if (rset.next()) {
78             String lastName = rset.getString(1);
79             String mi = rset.getString(2);
80             String firstName = rset.getString(3);
81             String title = rset.getString(4);
82             String grade = rset.getString(5);
83
84             // Display result in a label
85             lblStatus.setText(firstName + " " + mi +
86                 " " + lastName + "'s grade on course " + title + " is " +
87                 grade);
88         } else {
89             lblStatus.setText("Not found");
90         }
91     }
92     catch (SQLException ex) {
93         ex.printStackTrace();
94     }
95 }
96 }

```

initializeDB() 方法 (第 42 ~ 62 行) 装载 MySQL 驱动程序 (第 45 行), 连接主机

liang.armstrong.edu 上的 MySQL 数据库 (第 50 ~ 51 行), 并且创建一条语句 (第 57 行)。

【注意】 这个程序中有一个安全漏洞。如果在 SSN 字段中输入 `1' or true or '1`, 就会得到第一个学生的成绩, 这是因为查询字符串现在变成了:

```
select firstName, mi, lastName, title, grade
from Student, Enrollment, Course
where Student.ssn = '1' or true or '1' and
      Enrollment.courseId = ' ' and
      Enrollment.courseId = Course.courseId and
      Enrollment.ssn = Student.ssn;
```

可以使用 `PreparedStatement` 接口来避免这个问题。在下一节中我们将讨论到。

复习题

- 32.14 使用 Java 开发数据库应用程序的优势是什么?
- 32.15 描述下面的 JDBC 接口: `Driver`、`Connection`、`Statement` 和 `ResultSet`。
- 32.16 如何加载一个 JDBC 驱动程序? MySQL、Access 和 Oracle 的驱动程序类是什么?
- 32.17 如何创建一个数据库的连接? MySQL、Access 和 Oracle 的 URL 是什么?
- 32.18 如何创建一个 `Statement` 对象? 如何执行一个 SQL 语句?
- 32.19 如何获取 `ResultSet` 中的值?
- 32.20 JDBC 能自动地提交事务吗? 如何将自动提交模式设为 `false`?

32.5 PreparedStatement

 **要点提示:** `PreparedStatement` 可以创建参数化的 SQL 语句。

一旦建立了一个到特定数据库的连接, 就可以用这个连接把程序的 SQL 语句发送到数据库。`Statement` 接口用于执行不含参数的静态 SQL 语句。`PreparedStatement` 接口继承自 `Statement` 接口, 用于执行含有或不含参数的预编译的 SQL 语句。由于 SQL 语句是预编译的, 所以重复执行它们时效率较高。

`PreparedStatement` 对象是用 `Connection` 接口中的 `prepareStatement` 方法创建的。例如, 下面的代码为 SQL 语句 `insert` 创建一个 `PreparedStatement` 对象:

```
PreparedStatement preparedStatement = connection.prepareStatement
("insert into Student (firstName, mi, lastName) " +
 "values (?, ?, ?)");
```

这条 `insert` 语句有三个问号用作参数的占位符, 它们表示表 `Student` 中一条记录的 `firstName`、`mi` 和 `lastName` 的值。

作为 `Statement` 接口的子接口, `PreparedStatement` 接口继承了 `Statement` 接口中定义的所有方法。它还提供在 `PreparedStatement` 对象中设置参数的方法。这些方法用来在执行语句或过程之前设置参数的值。一般的, 设置的方法有如下的名字和签名:

```
setX(int parameterIndex, X value);
```

其中 `X` 是参数的类型, `parameterIndex` 是语句中参数的下标。下标从 1 开始。例如, 方法 `setString(int parameterIndex, String value)` 把一个 `String` 类型的值设置给指定参数。

下面的语句将参数 `"Jack"`、`"A"`、`"Ryan"` 传递给 `preparedStatement` 对象中 `firstName`、`mi` 和 `lastName` 的占位符:

```
preparedStatement.setString(1, "Jack");
preparedStatement.setString(2, "A");
preparedStatement.setString(3, "Ryan");
```


设置参数以后, 通过在 SELECT 语句中调用方法 `executeQuery()` 以及在 DDL 或更新语句中调用方法 `executeUpdate()`, 就可以执行预备好的语句。

`executeQuery()` 和 `executeUpdate()` 方法与定义在 `Statement` 接口中的这两个方法相似, 只是它们没有参数, 因为在创建 `PreparedStatement` 对象时, 已经在 `preparedStatement` 方法中指定了 SQL 语句。

使用预备的 SQL 语句, 程序清单 32-2 可以改进为程序清单 32-3。

程序清单 32-3 FindGradeUsingPreparedStatement.java

```

1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.scene.control.Label;
5  import javafx.scene.control.TextField;
6  import javafx.scene.layout.HBox;
7  import javafx.scene.layout.VBox;
8  import javafx.stage.Stage;
9  import java.sql.*;
10
11 public class FindGradeUsingPreparedStatement extends Application {
12     // PreparedStatement for executing queries
13     private PreparedStatement preparedStatement;
14     private TextField tfSSN = new TextField();
15     private TextField tfCourseId = new TextField();
16     private Label lblStatus = new Label();
17
18     @Override // Override the start method in the Application class
19     public void start(Stage primaryStage) {
20         // Initialize database connection and create a Statement object
21         initializeDB();
22
23         Button btShowGrade = new Button("Show Grade");
24         HBox hBox = new HBox(5);
25         hBox.getChildren().addAll(new Label("SSN"), tfSSN,
26             new Label("Course ID"), tfCourseId, (btShowGrade));
27
28         VBox vBox = new VBox(10);
29         vBox.getChildren().addAll(hBox, lblStatus);
30
31         tfSSN.setPrefColumnCount(6);
32         tfCourseId.setPrefColumnCount(6);
33         btShowGrade.setOnAction(e -> showGrade());
34
35         // Create a scene and place it in the stage
36         Scene scene = new Scene(vBox, 420, 80);
37         primaryStage.setTitle("FindGrade"); // Set the stage title
38         primaryStage.setScene(scene); // Place the scene in the stage
39         primaryStage.show(); // Display the stage
40     }
41
42     private void initializeDB() {
43         try {
44             // Load the JDBC driver
45             Class.forName("com.mysql.jdbc.Driver");
46             // Class.forName("oracle.jdbc.driver.OracleDriver");
47             System.out.println("Driver loaded");
48
49             // Establish a connection
50             Connection connection = DriverManager.getConnection
51                 ("jdbc:mysql://localhost/javabook", "scott", "tiger");
52             // ("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl",
53             //     "scott", "tiger");

```

```

54     System.out.println("Database connected");
55
56     String queryString = "select firstName, mi, " +
57         "lastName, title, grade from Student, Enrollment, Course " +
58         "where Student.ssn = ? and Enrollment.courseId = ? " +
59         "and Enrollment.courseId = Course.courseId";
60
61     // Create a statement
62     PreparedStatement preparedStatement = connection.prepareStatement(queryString);
63 }
64 catch (Exception ex) {
65     ex.printStackTrace();
66 }
67 }
68
69 private void showGrade() {
70     String ssn = tfSSN.getText();
71     String courseId = tfCourseId.getText();
72     try {
73         preparedStatement.setString(1, ssn);
74         preparedStatement.setString(2, courseId);
75         ResultSet rset = preparedStatement.executeQuery();
76
77         if (rset.next()) {
78             String lastName = rset.getString(1);
79             String mi = rset.getString(2);
80             String firstName = rset.getString(3);
81             String title = rset.getString(4);
82             String grade = rset.getString(5);
83
84             // Display result in a label
85             lblStatus.setText(firstName + " " + mi +
86                 " " + lastName + "'s grade on course " + title + " is " +
87                 grade);
88         } else {
89             lblStatus.setText("Not found");
90         }
91     }
92     catch (SQLException ex) {
93         ex.printStackTrace();
94     }
95 }
96 }

```

除了使用预备好的语句动态地设置参数外，本例与程序清单 32-2 执行了完全相同的操作。本例中的代码与程序清单 32-2 中的代码几乎完全相同，新代码采用加灰色突出背景显示。

第 56 ~ 59 行用 `ssn` 和 `courseId` 作为参数定义了一个预备好的查询字符串。第 62 行得到一个预备好的 SQL 语句。在执行这个查询之前，第 73 ~ 74 行将 `ssn` 和 `courseId` 的实际值设置到参数中。第 75 行执行预备好的语句。

✓ 复习题

- 32.21 描述预备语句的概念。如何创建 `PreparedStatement` 的一个实例？如何执行一个 `PreparedStatement` 对象？如何在 `PreparedStatement` 中设置参数值？
- 32.22 使用预备语句的好处是什么？

32.6 CallableStatement

 **要点提示：** `CallableStatement` 可以执行 SQL 存储过程。

`CallableStatement` 接口是为执行 SQL 存储过程而设计的。这个进程可能会有 `IN`、`OUT`

或 IN OUT 参数。当调用过程时，参数 IN 接收传递给过程的值。在进程结束后，参数 OUT 返回一个值，但是当调用过程时，它不包含任何值。当过程被调用时，IN OUT 参数包含传递给过程的值，在它完成之后返回一个值。例如，在 Oracle PL/SQL 的如下过程中有 IN 参数 p1、OUT 参数 p2 以及 IN OUT 参数 p3。

```
create or replace procedure sampleProcedure
  (p1 in varchar, p2 out number, p3 in out integer) is
begin
  /* do something */
end sampleProcedure;
/
```

{ } 注意：存储过程的语法是和特定厂商相关的。这里使用 Oracle 和 MySQL 演示本书中的存储进程。

可以使用 Connection 接口中的 prepareCall(String call) 创建 CallableStatement 对象。例如，下面的代码为进程 sampleProcedure 创建一个 Connection connection 上的 Callable Statement pstmt。

```
CallableStatement callableStatement = connection.prepareCall(
  "{call sampleProcedure(?, ?, ?)}");
```

{call sampleProcedure(?,?,...)} 指的是 SQL 转义语法，它通知驱动程序其中的代码应该被不同处理。驱动程序解析转义语法，并且将它翻译成数据库可以理解的代码。本例中，sampleProcedure 是一个 Oracle 过程。这个调用被翻译成字符串 begin sampleProcedure(?,?,?);end，然后传给 Oracle 数据库来执行。

既可以调用过程，也可以调用函数。为函数创建一个 SQL 的 callable 语句的语法如下所示：

```
{? = call functionName(?, ?, ...)}
```

CallableStatement 继承了 PreparedStatement。此外，CallableStatement 接口提供注册 OUT 参数的方法以及从 OUT 参数获取值的方法。

在调用 SQL 进程之前，需要使用合适的 setter 方法将值传给 IN 和 IN OUT 参数，使用 registerOutParameter 来注册 OUT 和 IN OUT 参数。例如，在调用进程 sampleProcedure 之前，下面的语句将值传给参数 p1(IN) 和 p3(IN OUT)，并注册参数 p2(OUT) 和 p3(IN OUT)：

```
callableStatement.setString(1, "Dallas"); // Set Dallas to p1
callableStatement.setLong(3, 1); // Set 1 to p3
// Register OUT parameters
callableStatement.registerOutParameter(2, java.sql.Types.DOUBLE);
callableStatement.registerOutParameter(3, java.sql.Types.INTEGER);
```

可以使用 execute() 或 executeUpdate() 按照 SQL 语句的类型执行进程，然后使用 getter 方法获取来自 OUT 参数的值。例如，下一条语句从参数 p2 和 p3 获取值。

```
double d = callableStatement.getDouble(2);
int i = callableStatement.getInt(3);
```

让我们定义一个 MySQL 函数，返回表中和 Student 表中指定的 firstName 和 lastName 相匹配的记录个数。

```
/* For the callable statement example. Use MySQL version 5 */
drop function if exists studentFound;

delimiter //

create function studentFound(first varchar(20), last varchar(20))
```

```

    returns int
begin
    declare result int;

    select count(*) into result
    from Student
    where Student.firstName = first and
           Student.lastName = last;

    return result;
end;
//

delimiter ;
/* Please note that there is a space between delimiter and ; */

```

如果使用 Oracle 数据库, 函数可以如下定义:

```

create or replace function studentFound
(first varchar2, last varchar2)
/* Do not name firstName and lastName. */
return number is
numberOfSelectedRows number := 0;
begin
    select count(*) into numberOfSelectedRows
    from Student
    where Student.firstName = first and
           Student.lastName = last;

    return numberOfSelectedRows;
end studentFound;
/

```

假设数据库中已经创建了 studentFound 函数。程序清单 32-4 给出一个使用 callable 语句测试该函数的例子。

程序清单 32-4 TestCallableStatement.java

```

1  import java.sql.*;
2
3  public class TestCallableStatement {
4      /** Creates new form TestTableEditor */
5      public static void main(String[] args) throws Exception {
6          Class.forName("com.mysql.jdbc.Driver");
7          Connection connection = DriverManager.getConnection(
8              "jdbc:mysql://localhost/javabook",
9              "scott", "tiger");
10         // Connection connection = DriverManager.getConnection(
11         //     ("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl",
12         //     "scott", "tiger");
13
14         // Create a callable statement
15         CallableStatement callableStatement = connection.prepareCall(
16             "{? = call studentFound(?, ?)}");
17
18         java.util.Scanner input = new java.util.Scanner(System.in);
19         System.out.print("Enter student's first name: ");
20         String firstName = input.nextLine();
21         System.out.print("Enter student's last name: ");
22         String lastName = input.nextLine();
23
24         callableStatement.setString(2, firstName);
25         callableStatement.setString(3, lastName);
26         callableStatement.registerOutParameter(1, Types.INTEGER);
27         callableStatement.execute();

```

```

28
29     if (callableStatement.getInt(1) >= 1)
30         System.out.println(firstName + " " + lastName +
31             " is in the database");
32     else
33         System.out.println(firstName + " " + lastName +
34             " is not in the database");
35 }
36 }

```

Enter student's first name: Jacob

Enter student's last name: Smith

Jacob Smith is in the database

Enter student's first name: John

Enter student's last name: Smith

John Smith is not in the database

该程序加载 MySQL 驱动程序 (第 6 行), 连接到 MySQL 数据库 (第 7 ~ 9 行), 并且创建一个执行函数 `studentFound` 的 `callable` 语句 (第 15 ~ 16 行)。

函数的第一个参数是返回值, 第二个和第三个参数对应的是名和姓。在执行 `callable` 语句之前, 程序设置名和姓 (第 24 ~ 25 行) 并注册 OUT 参数 (第 26 行)。该语句在第 27 行执行。

在第 29 行获取函数的返回值。如果这个值大于或等于 1, 那么就能找到表格中有特定名和姓的学生。

✓ 复习题

32.23 描述 `callable` 语句。如何创建 `CallableStatement` 的一个实例? 如何执行一个 `CallableStatement` 对象? 如何在 `CallableStatement` 中注册 OUT 参数值?

32.7 获取元数据

要点提示: 可以使用 `DatabaseMetaData` 接口来获取数据库的元数据, 例如数据库 URL、用户名、JDBC 驱动程序名称等。`ResultSetMetaData` 接口可以用于获取到结果集合的元数据, 例如表的列数和列名等。

JDBC 提供 `DatabaseMetaData` 接口, 可用来获取数据库范围的信息, 还提供 `ResultSetMetaData` 接口, 用于获取特定的 `ResultSet` 的信息。

32.7.1 数据库元数据

`Connection` 接口用于建立与数据库的连接。SQL 语句的执行和结果的返回是在一个连接上下文中进行的。连接还提供对数据库元数据信息的访问, 该信息描述了数据库的能力、支持的 SQL 语法、存储过程, 等等。要得到数据库的一个 `DatabaseMetaData` 实例, 可以使用 `Connection` 对象的 `getMetaData` 方法, 如下所示:

```
DatabaseMetaData dbMetaData = connection.getMetaData();
```

如果你的程序连接到一个本地 MySQL 数据库, 程序清单 32-5 显示了数据库的信息, 如图 32-24 所示。

程序清单 32-5 TestDatabaseMetaData.java

```

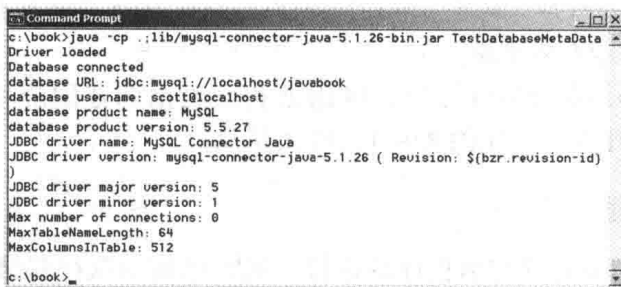
1 import java.sql.*;
2

```

```

3 public class TestDatabaseMetaData {
4     public static void main(String[] args)
5         throws SQLException, ClassNotFoundException {
6         // Load the JDBC driver
7         Class.forName("com.mysql.jdbc.Driver");
8         System.out.println("Driver loaded");
9
10        // Connect to a database
11        Connection connection = DriverManager.getConnection
12            ("jdbc:mysql://localhost/javabook", "scott", "tiger");
13        System.out.println("Database connected");
14
15        DatabaseMetaData dbMetaData = connection.getMetaData();
16        System.out.println("database URL: " + dbMetaData.getURL());
17        System.out.println("database username: " +
18            dbMetaData.getUserName());
19        System.out.println("database product name: " +
20            dbMetaData.getDatabaseProductName());
21        System.out.println("database product version: " +
22            dbMetaData.getDatabaseProductVersion());
23        System.out.println("JDBC driver name: " +
24            dbMetaData.getDriverName());
25        System.out.println("JDBC driver version: " +
26            dbMetaData.getDriverVersion());
27        System.out.println("JDBC driver major version: " +
28            dbMetaData.getDriverMajorVersion());
29        System.out.println("JDBC driver minor version: " +
30            dbMetaData.getDriverMinorVersion());
31        System.out.println("Max number of connections: " +
32            dbMetaData.getMaxConnections());
33        System.out.println("MaxTableNameLength: " +
34            dbMetaData.getMaxTableNameLength());
35        System.out.println("MaxColumnsInTable: " +
36            dbMetaData.getMaxColumnsInTable());
37
38        // Close the connection
39        connection.close();
40    }
41 }

```



```

c:\book>java -cp .\lib\mysql-connector-java-5.1.26-bin.jar TestDatabaseMetaData
Driver loaded
Database connected
database URL: jdbc:mysql://localhost/javabook
database username: scott@localhost
database product name: MySQL
database product version: 5.5.27
JDBC driver name: MySQL Connector Java
JDBC driver version: mysql-connector-java-5.1.26 ( Revision: $(bazaar.revision-id)
)
JDBC driver major version: 5
JDBC driver minor version: 1
Max number of connections: 0
MaxTableNameLength: 64
MaxColumnsInTable: 512
c:\book>

```

图 33-24 可以使用 DatabaseMetaData 接口获取数据库信息

32.7.2 获取数据库表

使用 `getTables` 方法通过数据库元数据可以确定数据库中的表格。程序清单 32-6 显示了在本地 MySQL 数据库 `javabook` 中的所有用户表。图 32-25 显示了该程序的示例输出。

程序清单 32-6 FindUserTables.java

```

1 import java.sql.*;
2
3 public class FindUserTables {

```

```

4 public static void main(String[] args)
5     throws SQLException, ClassNotFoundException {
6     // Load the JDBC driver
7     Class.forName("com.mysql.jdbc.Driver");
8     System.out.println("Driver loaded");
9
10    // Connect to a database
11    Connection connection = DriverManager.getConnection
12        ("jdbc:mysql://localhost/javabook", "scott", "tiger");
13    System.out.println("Database connected");
14
15    DatabaseMetaData dbMetaData = connection.getMetaData();
16
17    ResultSet rsTables = dbMetaData.getTables(null, null, null,
18        new String[] {"TABLE"});
19    System.out.print("User tables: ");
20    while (rsTables.next())
21        System.out.print(rsTables.getString("TABLE_NAME") + " ");
22
23    // Close the connection
24    connection.close();
25 }
26 }

```

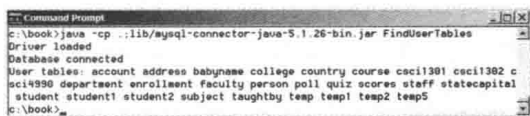


图 32-25 可以得到数据库中的所有表

第 17 行使用 `getTables` 方法在结果集中获取表信息。结果集中的一列是 `TABLE_NAME`。第 21 行从结果集的列中获取表名。

32.7.3 结果集元数据

`ResultSetMetaData` 接口描述属于结果集的信息。`ResultSetMetaData` 对象能够用于在结果集 `ResultSet` 中找出关于列的类型和属性的信息。要得到 `ResultSetMetaData` 的一个实例，可在结果集上使用 `getMetaData` 方法，如下所示：

```
ResultSetMetaData rsMetaData = resultSet.getMetaData();
```

使用 `getColumnCount()` 方法可以在结果中求得列的数目，使用 `getColumnName(int)` 方法可以求得列名。例如，程序清单 32-7 显示从 SQL `SELECT` 语句 `select * from Enrollment` 得到的所有列名和内容，输出结果如图 32-26 所示。

程序清单 32-7 TestResultSetMetaData.java

```

1 import java.sql.*;
2
3 public class TestResultSetMetaData {
4     public static void main(String[] args)
5         throws SQLException, ClassNotFoundException {
6         // Load the JDBC driver
7         Class.forName("com.mysql.jdbc.Driver");
8         System.out.println("Driver loaded");
9
10        // Connect to a database
11        Connection connection = DriverManager.getConnection
12            ("jdbc:mysql://localhost/javabook", "scott", "tiger");

```



```

12      ("jdbc:mysql://localhost/javabook", "scott", "tiger");
13      System.out.println("Database connected");
14
15      // Create a statement
16      Statement statement = connection.createStatement();
17
18      // Execute a statement
19      ResultSet resultSet = statement.executeQuery
20      ("select * from Enrollment");
21
22      ResultSetMetaData rsMetaData = resultSet.getMetaData();
23      for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
24          System.out.printf("%-12s\t", rsMetaData.getColumnName(i));
25      System.out.println();
26
27      // Iterate through the result and print the students' names
28      while (resultSet.next()) {
29          for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
30              System.out.printf("%-12s\t", resultSet.getObject(i));
31          System.out.println();
32      }
33
34      // Close the connection
35      connection.close();
36  }
37  }

```

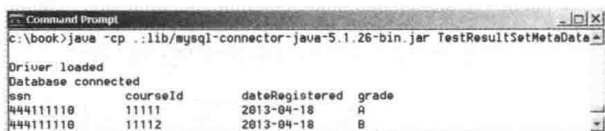


图 32-26 利用 ResultSetMetaData 接口可以获取结果集的信息

复习题

- 32.24 DatabaseMetaData 的作用是什么？描述 DatabaseMetaData 中的方法。如何获得 DatabaseMetaData 的一个实例？
- 32.25 ResultSetMetaData 的作用是什么？描述 ResultSetMetaData 中的方法。如何获得 ResultSetMetaData 的一个实例？
- 32.26 如何在结果集中求得列的数目？如何在结果集中求得列名？

关键术语

candidate key (候选键)	integrity constraint (完整性约束)
database system (数据库系统)	primary key (主键)
domain constraint (域约束)	relational database (关系数据库)
foreign key (外键)	Structured Query Language (SQL, 结构化查询语言)
foreign key constraint (外键约束)	superkey (超键)

本章小结

- 本章介绍了数据库系统、关系数据库、关系数据模型、数据完整性和 SQL 的概念，还介绍了如何使用 Java 开发数据库应用程序。
- 用于开发 Java 数据库应用程序的 Java API 称为 JDBC。JDBC 给 Java 编程人员提供了一个访问和操作关系数据库的统一接口。

3. JDBC API 由接口和类组成。这些类和接口用于建立与数据库的连接、把 SQL 语句发送到数据库、处理 SQL 语句的结果，以及获取数据库的元数据。
4. 因为 JDBC 驱动程序起着接口的作用，它使 JDBC 与具体数据库之间的通信灵活方便，所以，JDBC 驱动程序是与具体数据库相关的。JDBC-ODBC 桥式驱动程序包含在 JDK 中，用来支持通过 ODBC 驱动程序访问数据库的 Java 程序。如果使用的驱动程序不是 JDBC-ODBC 桥式驱动程序，在运行程序前必须确保它在类路径 (classpath) 上。
5. 使用 Java 开发任何数据库应用程序都需要 4 个主要接口：Driver、Connection、Statement 和 ResultSet。这些接口定义了使用 SQL 数据库访问的一般架构。JDBC 驱动程序开发商提供了它们的实现。
6. JDBC 应用程序使用 Driver 接口加载一个合适的驱动程序，使用 Connection 接口连接数据库，使用 Statement 接口创建并执行 SQL 语句，如果语句返回结果，使用 ResultSet 接口处理结果。
7. PreparedStatement 接口是为执行带参数的动态 SQL 语句而设计的。为了提高重复执行的效率，对这些 SQL 语句进行了预编译。
8. 数据库的元数据描述数据库本身的信息。JDBC 为获取数据库范围的信息提供了 DatabaseMetaData 接口，为得到具体结果集 ResultSet 的信息提供了 ResultSetMetaData 接口。

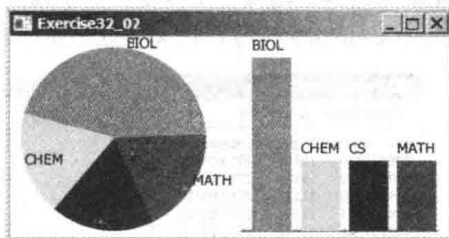
测试题

回答位于网址 www.cs.armstrong.edu/liang/intro10e/quiz.html 的本章测试题。

编程练习题

- *32.1 (访问并更新表 Staff) 编写一个程序，浏览、插入和更新存储在一个数据库中的职员信息，如图 32-27a 所示。View 按钮用于显示具有指定 ID 的记录。Insert 按钮插入一条新的记录。Update 按钮更新一条指定 ID 的记录。按如下方式创建职工表 Staff：

a)



b)

图 32-27 a) 程序能够浏览、插入和更新职工信息；b) PieChart 和 BarChart 组件显示从数据模块中获取的查询数据

```
create table Staff (
    id char(9) not null,
    lastName varchar(15),
    firstName varchar(15),
    mi char(1),
    address varchar(20),
    city varchar(20),
    state char(2),
    telephone char(10),
    email varchar(40),
    primary key (id)
);
```

- **32.2 (可视化数据) 编写一个程序，在一个饼状图和条状图中显示每个系的学生数目，如图 32-27b 所

示。每个系的学生数目可以使用以下 SQL 语句从 Student 表中获得 (参见图 32-4):

```
select deptId, count(*)
from Student
where deptId is not null
group by deptId;
```

*32.3 (连接对话框) 开发一个名为 DBConnectionPane 的 BorderPane 的子类, 使用户能够选择或输入 JDBC 驱动程序和 URL, 并且可以输入用户名和口令, 如图 32-28 所示。当单击 Connect to DB 按钮时, 与数据库关联的 Connection 对象存储到 connection 属性中, 然后使用 getConnection() 方法返回连接。

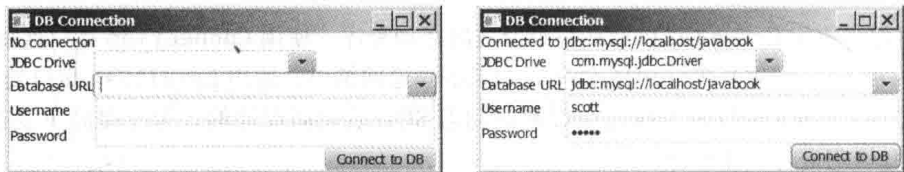


图 32-28 DBConnectionPane 组件使得用户可以输入数据库信息

*32.4 (查找成绩) 程序清单 32-2 给出了一个程序, 可以查找学生的指定课程的成绩。改写该程序, 查找指定学生的所有成绩, 如图 32-29 所示。

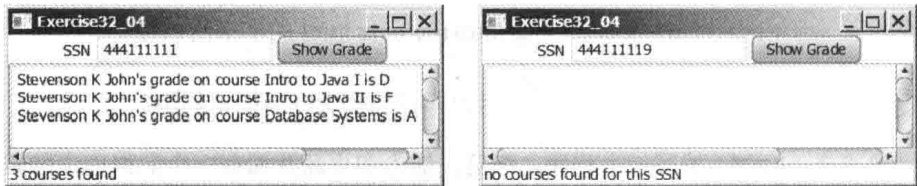


图 32-29 程序显示指定学生的课程成绩

*32.5 (显示表的内容) 编写一个程序, 显示指定表的内容。如图 32-30a 所示, 输入一个表名, 然后单击 Show Contents 按钮, 在文本域中显示表的内容。

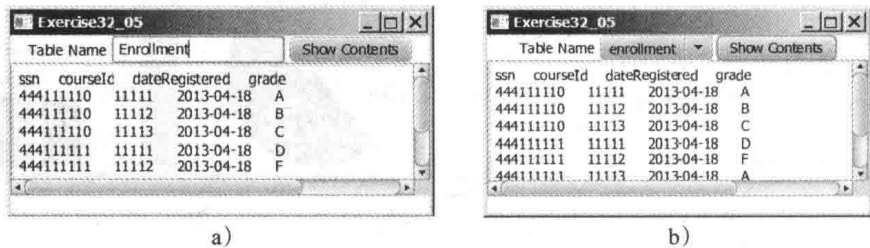


图 32-30 a) 输入表名以显示表的内容; b) 从组合框中选择表名以显示它的内容

*32.6 (查找表并显示表的内容) 编写一个程序, 在组合框中填写表名, 如图 32-30b 所示。可以从组合框中选择一个表名, 然后在文本域中显示它的内容。

**32.7 (填充 Quiz 表) 创建一个名为 Quiz 的表, 如下所示:

```
create table Quiz(
    questionId int,
    question varchar(4000),
    choicea varchar(1000),
    choiceb varchar(1000),
    choicec varchar(1000),
    choiced varchar(1000),
    answer varchar(5));
```

Quiz 表存储了多选题。假设多选题以下面的格式存储在 www.cs.armstrong.edu/liang/data/Quiz.txt 的文本文件中:

```
1. question1
a. choice a
b. choice b
c. choice c
d. choice d
Answer:cd
```

```
2. question2
a. choice a
b. choice b
c. choice c
d. choice d
Answer:a
```

...

编写从文件读取数据的程序，然后将它存储在 Quiz 表中。

*32.8 (填充 Salary 表) 创建一个名为 Salary 的表，如下所示：

```
create table Salary(
    firstName varchar(100),
    lastName varchar(100),
    rank varchar(15),
    salary float);
```

从 <http://cs.armstrong.edu/liang/data/Salary.txt> 中获得薪水的数据，并将其填充到数据库的 Salary 表中。

*32.9 (复制表) 假设数据库中包含了一个学生表格，如下所示：

```
create table Student1 (
    username varchar(50) not null,
    password varchar(50) not null,
    fullname varchar(200) not null,
    constraint pkStudent primary key (username)
);
```

创建一个名为 Student2 的表，如下所示：

```
create table Student2 (
    username varchar(50) not null,
    password varchar(50) not null,
    firstname varchar(100),
    lastname varchar(100),
    constraint pkStudent primary key (username)
);
```

完整的名字是 `firstname mi lastname` 或者 `firstname lastname` 的形式。例如，John K Smith 是全名。编写一个程序，复制表 Student1 到 Student2 中。你的任务是将 Student1 中的每条记录的全名分为 `firstname`、`mi` 和 `lastname`，并在 Student2 中存储一条新的记录。

*32.10 (记录未提交的习题情况) 以下三个表存储了关于学生、布置的习题，以及习题在 LiveLab 中提交情况的信息。LiveLab 是一个用于对编程练习题进行自动给分的系统。

```
create table AGSStudent (
    username varchar(50) not null,
    password varchar(50) not null,
    fullname varchar(200) not null,
    instructorEmail varchar(100) not null,
    constraint pkAGSStudent primary key (username)
);
```

```
create table ExerciseAssigned (
    instructorEmail varchar(100),
    exerciseName varchar(100),
    maxscore double default 10,
```

```

constraint pkCustomExercise primary key
(instructorEmail, exerciseName)
);

create table AGSLog (
  username varchar(50), /* This is the student's user name */
  exerciseName varchar(100), /* This is the exercise */
  score double default null,
  submitted bit default 0,
  constraint pkLog primary key (username, exerciseName)
);

```

AGSStudent 表存储了学生信息。ExerciseAssigned 表存储教师布置的习题。AGSLog 表存储了给分结果。当学生提交一个习题，一条记录保存在 AGSLog 表中。然而，如果学生没有提交习题，则在 AGSLog 表中没有记录。

编写一个程序，如果一个学生没有提交习题，则对该学生添加一条新的记录，并在 AGSLog 表的该学生下记录布置的习题信息。该记录的 score 和 submitted 字段应该值为 0。例如，如果在运行该程序前，表 AGSLog 中包含下列数据，则程序运行后，表 AGSLog 会包含如下新的记录。

AGSStudent

username	password	fullname	instructorEmail
abc	p1	John Roo	t@gmail.com
cde	p2	Yao Mi	c@gmail.com
wbc	p3	F3	t@gmail.com

ExerciseAssigned

instructorEmail	exerciseName	maxScore
t@gmail.com	e1	10
t@gmail.com	e2	10
c@gmail.com	e1	4
c@gmail.com	e4	20

AGSLog

username	exerciseName	score	submitted
abc	e1	9	1
wbc	e2	7	1

AGSLog after the program runs

username	exerciseName	score	submitted
abc	e1	9	1
wbc	e2	7	1
abc	e2	0	0
wbc	e1	0	0
cde	e1	0	0
cde	e4	0	0

*32.11 (婴儿姓名) 创建以下表格：

```

create table Babyname (
  year integer,
  name varchar(50),
  gender char(1),
  count integer,
  constraint pkBabyname primary key (year, name, gender)
);

```

婴儿姓名的排行信息在编程练习题 12.31 中描述。编写一个程序，从以下 URL 读取数据并存储到 Babyname 表中。

<http://www.cs.armstrong.edu/liang/data/babynamesranking2001.txt>

...

<http://www.cs.armstrong.edu/liang/data/babynamesranking2010.txt>

JavaServer Faces

【】 教学目标

- 解释 JSF 是什么 (33.1 节)。
- 在 NetBeans 中创建一个 JSF 项目 (33.2.1 节)。
- 创建一个 JSF 页面 (33.2.2 节)。
- 创建一个 JSF 管理的 bean (33.2.3 节)。
- 在一个 facelet 中使用 JSF 表达式 (33.2.4 节)。
- 使用 JSF GUI 组件 (33.3 节)。
- 从一个表单中得到和处理输入 (33.4 节)。
- 使用 JSF 开发一个计算器 (33.5 节)。
- 在应用程序、会话、视图和请求范围内跟踪会话 (33.6 节)。
- 使用 JSF 验证器来验证输入 (33.7 节)。
- 将数据库与 facelet 绑定 (33.8 节)。
- 从当前页面打开一个新的 JSF 页面 (33.9 节)。

33.1 引言

🔑 **要点提示:** JavaServer Faces (JSF) 是使用 Java 开发服务器端 Web 应用的一项新技术。

JSF 使得 Java 代码可以完全与 HTML 分离。可以通过在一个页面中组装可重用的 UI 组件,然后将这些组件与 Java 程序连接,以及将客户端产生的事件连接到服务器端的事件处理器,来快速构建 Web 应用,使用 JSF 开发的应用易于调试和维护。

【】 **注意:** 本章介绍 JSF2.2 这一 JavaServer Faces 的最新标准。你需要有 XHTML (eXtensible HyperText Markup Language) 和 CSS (Cascading Style Sheet) 的知识来开始本章的学习。关于 XHTML 和 CSS 的相关信息,参见配套网站上的补充材料 V.A 和 V.B。

【】 **警告:** 本章中的示例和练习使用 NetBeans 7.4、GlassFish4、JSF2.2 以及 J2EE7 进行过测试。你需要使用 NetBeans 7.4 或者以上版本和 GlassFish4、JSF2.2 以及 J2EE 来开发 JSF 项目。

33.2 开始使用 JSF

🔑 **要点提示:** NetBeans 是开发 JSF 应用的一个高效工具。

我们以一个简单的示例开始,演示使用 NetBeans 开发 JSF 项目的基础。该示例用于显示服务器端的日期和时间,如图 33-1 所示。

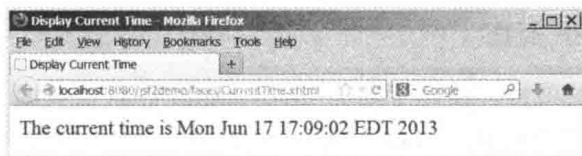


图 33-1 该应用显示服务器端的日期和时间

33.2.1 创建一个 JSF 项目

下面是创建应用的步骤。

步骤 1：选择 File → New Project 来显示一个 New Project 对话框。在该对话框中，在 Categories 面板选择 Java Web，在 Project 面板中选择 Web Application。点击 Next 显示 New Web Application 对话框。

在 New Web Application 对话框中，输入和选择以下字段，如图 33-2a 所示：

Project Name: jsf2demo

Project Location: c:\book

步骤 2：点击 Next 显示用于选择服务器和设置的对话框。选择以下字段，如图 33-2b 所示。（注意：可以使用任何支持 JavaEE 7 的服务器，比如 GlassFish 4.x。）

Server: GlassFish 4

Java EE Version: Java EE 7 Web

步骤 3：点击 Next 显示用于选择框架的对话框，如图 33-3 所示。勾选 JavaServer Faces 和 JSF 2.2 作为 Server Library（服务器库）。点击 Finish 来创建项目，如图 33-4 所示。

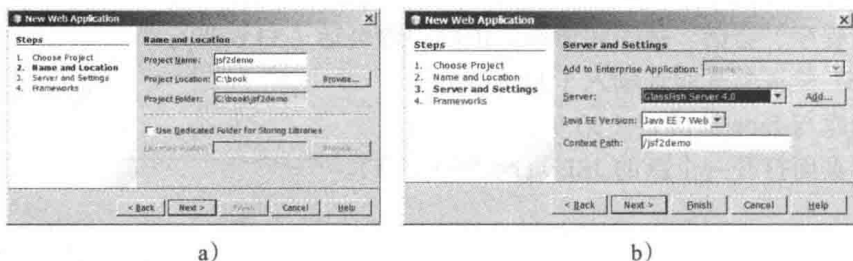


图 33-2 New Web Application 对话框可以用于创建一个新的 Web 项目

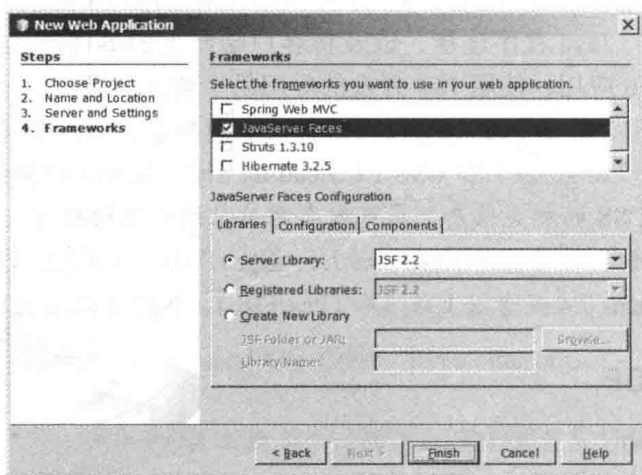


图 33-3 勾选 JavaServer Faces 和 JSF 2.2，创建一个新的 Web 项目

33.2.2 一个基本的 JSF 页面

刚才创建了一个新的项目，使用了名为 index.xhtml 的默认页面，如图 33-4 所示。该页面称为一个 facelet，它混合使用了 JSF 标签和 XHTML 标签。程序清单 33-1 列出了 index.xhtml 的内容。

程序清单 33-1 index.xhtml1

```
1 <?xml version='1.0' encoding='UTF-8' ?>
2 <!-- index.xhtml1 -->
3 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
4   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
5 <html xmlns="http://www.w3.org/1999/xhtml"
6       xmlns:h="http://xmlns.jcp.org/jsf/html">
7   <h:head>
8       <title>Facelet Title</title>
9   </h:head>
10  <h:body>
11      Hello from Facelets
12  </h:body>
13 </html>
```

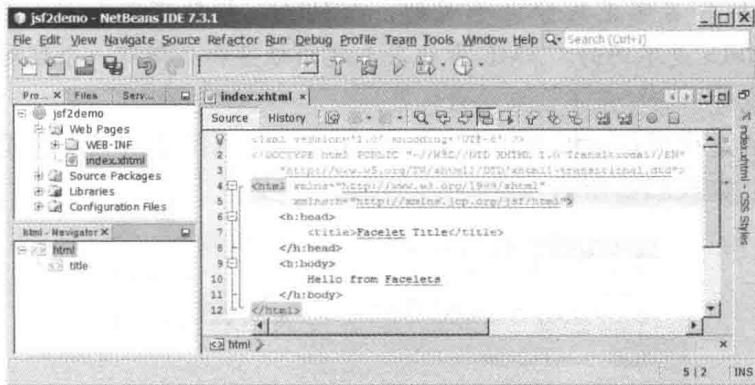


图 33-4 在一个新的 Web 项目中创建了一个默认的 JSF 页面

第 1 行是一个 XML 的声明，说明该文档遵循 XML 版本 1.0，并且使用 UTF-8 编码。该声明是可选的，但是使用它是一个好的做法。没有声明的文档可能被认为是一个不同的版本，这将可能导致错误。如果给出了一个 XML 声明，它需要出现在文档的第一行。因为 XML 处理器从第一行中得到关于文档的信息，从而可以正确地进行处理。

第 2 行是一个注释，用于对文件中的内容进行记载。XML 注释通常以 `<!--` 开始，以 `-->` 结束。

第 3、4 行给定该文档使用的 XHTML 版本。可以让 Web 浏览器对该文档的语法进行验证。

XML 文档由标签描述的元素组成。元素包含在起始标签和结束标签之间。XML 元素以一种树状的层次进行组织。元素可以包含子元素，但一个 XML 文档只有一个根元素。所有的元素必须被包含在根标签中。XHTML 的根元素使用 `html` 标签定义（第 5 行）。

XML 中的每个标签必须采用一对起始标签和结束标签。起始标签以 `<` 开始，然后是标签名字，以 `>` 结束。结束标签和相应的起始标签一样，除了以 `</` 开始。`html` 的起始标签和结束标签分别是 `<html>` 和 `</html>`。

`html` 元素是根元素，包含了 XHTML 页面中的所有其他元素。起始标签 `<html>`（第 5 行和第 6 行）可能包含一个或者多个 `xmlns`（XML 名称空间）属性来指定文档中使用的元素的名称空间。名称空间类似 Java 的包。Java 的包用于组织类和解决命名冲突的问题。XHTML 名称空间用于组织标签和解决命名冲突的问题。如果一个元素在两个名称空间中定义了同样的名字，那么充分限定的标签名可以用于区分它们。

每个 `xmlns` 属性具有一个名字和一个值，由一个等号（=）分开。下面的声明（第 5 行）

```
xmlns="http://www.w3.org/1999/xhtml"
```

给出任何没有被限定的标签名都定义在默认的标准 XHTML 名称空间中。

以下声明 (第 6 行)

```
xmlns:h="http://xmlns.jcp.org/jsf/html"
```

使得定义在 JSF 标签库中的标签可以用于该文档中。这些标签必须具有一个前缀 h。

一个 html 元素包含一个头部分和一个体部分。h:head 元素 (第 7 ~ 9 行) 定义了一个 HTML 的 title 元素。标题通常显示在浏览器窗口的标题栏中。

h:body 元素定义了页面的内容。在这个简单的示例中, 它包含了一个显示在 Web 浏览器的字符串。

注意: XML 标签名字是区分大小写的, 而 HTML 标签不是如此。因此, 在 XML 中 <html> 和 <HTML> 是不同的。XML 中的每个起始标签和必须有一个配对的结束标签。然而, HTML 中的一些标签不需要结束标签。

现在, 可以通过在项目面板中右击 index.xhtml 并且选择 Run File 来显示页面了。页面在浏览器中显示, 如图 33-5 所示。



图 33-5 index.xhtml 显示在浏览器中

33.2.3 JSF 的受管 JavaBean

JSF 应用程序使用 Model-View-Controller (模型 - 视图 - 控制器, MVC) 架构开发, 这样将应用的数据 (包含在模型中) 和图形化表示 (视图) 分离开来。控制器是负责协调视图和模型之间交互的 JSF 框架。

JSF 中, facelet 是表示数据的视图。数据从 Java 对象处获取。对象使用 Java 类定义。JSF 中, facelet 访问的对象为 JavaBean 对象。JavaBean 类是一个简单的具有无参构造方法的 Java 类。JavaBean 可能具有属性。习惯上, 属性都定义有 getter 和 setter 方法。如果一个属性只具有 getter 方法, 该属性称为只读属性。如果一个属性只具有 setter 方法, 该属性称为只写属性。属性也不是必须定义为类中的一个数据域。

本节中我们的例子是开发一个显示当前时间的 JSF facelet。我们将开发一个 JavaBean, 具有一个以字符串返回当前时间的 getTime() 方法。facelet 将调用该方法得到当前时间。

这里是创建一个名为 TimeBean 的 JavaBean 步骤。

步骤 1: 右击项目结点 jsf2demo, 显示一个上下文菜单, 如图 33-6 所示。选择 New → JSF Managed Bean, 显示一个 New JSF Managed Bean 对话框, 如图 33-7 所示。(注意: 如果菜单中没有看到 JSF Managed Bean, 则在 JavaServer Faces 类别下面选择 Other 来进行定位)。

步骤 2: 输入和选择以下字段, 如图 33-7 所示:

```
Class Name: TimeBean
Package: jsf2demo
```

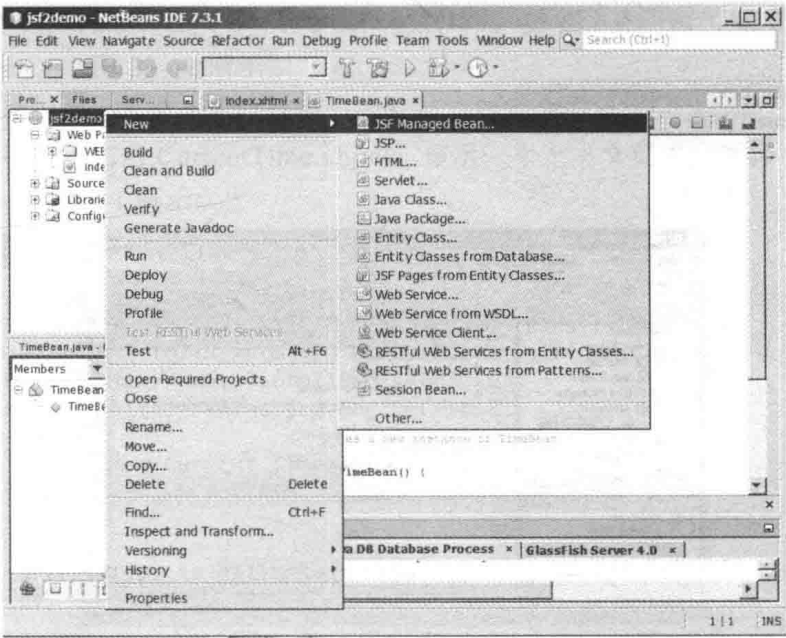


图 33-6 选择 JSF Managed Bean 来为 JSF 创建一个 JavaBean

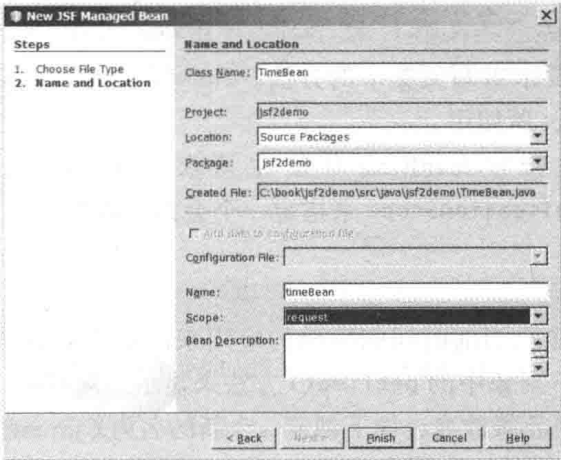


图 33-7 为 bean 指定名字、位置和范围

Name: timeBean
Scope: request

点击 Finish 来创建 TimeBean.java，如图 33-8 所示。
步骤 3：添加 getTime() 方法返回当前时间，如程序清单 33-2 所示。

程序清单 33-2 TimeBean.java

```
1 package jsf2demo;  
2  
3 import javax.inject.Named;  
4 import javax.enterprise.context.RequestScoped;  
5  
6 @Named  
7 @RequestScoped  
8 public class TimeBean {
```

```

9    public TimeBean() {
10   }
11
12   public String getTime() {
13       return new java.util.Date().toString();
14   }
15 }

```

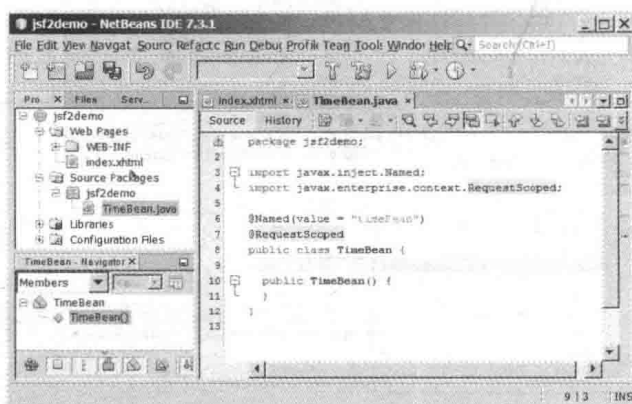


图 33-8 为 JSF 创建了一个 JavaBean

TimeBean 是一个具有 @Named 标注的 JavaBean，这意味着 JSF 框架将创建和管理应用中使用的 TimeBean 对象。你已经在第 11 章学习了使用 @Override 标注。用 @Override 标注告诉编译器被标注的方法要求重写父类中的方法。@Named 标注告诉编译器产生代码，从而使 bean 可以被 JSF facelet 所使用。

@RequestScope 标注指定 JavaBean 对象在请求范围内。也可以使用 @ViewScope、@SessionScope 或者 @ApplicationScope 来指定一个会话或者整个应用程序的范围等。

33.2.4 JSF 表达式

我们通过编写一个显示当前时间的简单的应用来演示 JSF 表达式。可以通过使用一个 JSF 表达式调用 TimeBean 对象中的 getTime() 方法来显示当前时间。

为了保持 index.xhtml 不被改变，如下创建一个新的名为 CurrentTime.xhtml 的 JSF 页面：

步骤 1：在项目面板中右击结点 jsf2demo，显示一个上下文菜单，选择 New → JSF Page，显示一个 New JSF Page 对话框，如图 33-9 所示。

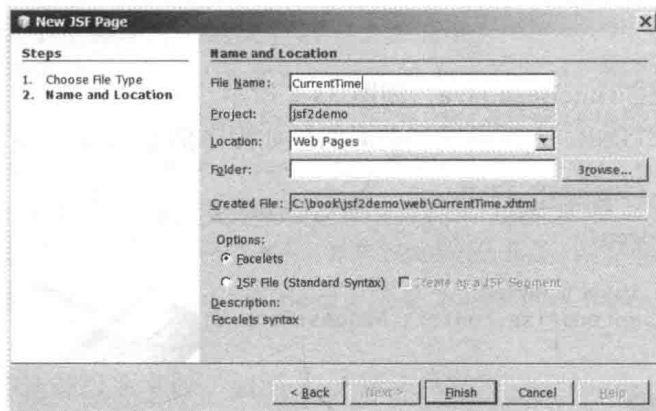


图 33-9 使用 New JSF Page 对话框创建一个 JSF 页面

步骤 2：在 File Name 字段中输入 CurrentTime，选择 Facelets 并点击 Finish 来生成 CurrentTime.xhtml，如图 33-10 所示。

步骤 3：添加一个 JSF 表达式以获得当前时间，如程序清单 33-3 所示。

步骤 4：在项目中右击 CurrentTime.xhtml，显示一个上下文菜单，选择 Run File 在浏览器中显示页面，如图 33-1 所示。

程序清单 33-3 CurrentTime.xhtml

```

1 <?xml version='1.0' encoding='UTF-8' ?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml"
5       xmlns:h="http://xmlns.jcp.org/jsf/html">
6   <h:head>
7     <title>Display Current Time</title>
8     <meta http-equiv="refresh" content="60" />
9   </h:head>
10  <h:body>
11    The current time is #{timeBean.time}
12  </h:body>
13 </html>

```



图 33-10 一个新的 JSF 页面 CurrentTime 被创建

第 8 行在 h:head 标签中定义了一个 meta 标签，告诉浏览器每 60 秒刷新一次。该行也可以如下编写：

```
<meta http-equiv="refresh" content="60"></ meta>
```

如果在起始标签和结束标签中没有内容，这样的元素称为空元素。在一个空元素中，数据一般在起始标签中给出。为简单起见，可以通过在起始标签的右括号之前放置一个斜杠关闭一个空元素，如第 8 行所示。

第 11 行使用一个 JSF 表达式 #{timeBean.time} 来获得当前时间。timeBean 是 TimeBean 类的一个对象。可以使用如下语法，在 @Named 标注中（程序清单 33-2 第 6 行）改变对象名字。

```
@Named(name = "anyObjectName")
```

默认的，对象名字为类的名字，但第一个字母为小写。

注意 time 是 JavaBean 属性，因为 getTime() 定义在 TimeBeans 中。JSF 表达式既可以使用属性名字，也可以使用方法来获得当前时间。因此以下两个表达式都是可以的：

```

#{timeBean.time}
#{timeBean.getTime()}

```

JSF 表达式的语法如下：

```
{expression}
```

JSF 表达式将 JavaBean 对象和 facelet 绑定。在本章下面的示例中将看到更加多的 JSF 表达式的应用。

复习题

- 33.1 什么是 JSF ?
- 33.2 如何在 NetBeans 中创建一个 JSF 项目?
- 33.3 如何在一个 JSF 项目中创建一个 JSF 页面?
- 33.4 什么是 facelet?
- 33.5 facelet 的文件后缀名是什么?
- 33.6 什么是受管 bean?
- 33.7 @Named 标注用于什么?
- 33.8 @RequestScope 标注用于什么?

33.3 JSF GUI 组件

要点提示：JSF 提供了许多元素用于显示 GUI 组件。

表 33-1 列出了一些常用的元素。以 h 为前缀的标签位于 JSF HTML 标签库中。以 f 为前缀的标签位于 JSF Core 标签库中。

表 33-1 JSF GUI 表单元素

JSF 标签	描述
h:form	插入一个 XHTML 表单到页面中
h:panelGroup	类似于 JavaFX 的 FlowPane
h:panelGrid	类似于 JavaFX 的 GridPane
h:inputText	显示一个文本框用于输入
h:outputText	显示一个文本框用于显示输出
h:inputTextArea	显示一个文本区域用于输入
h:inputSecret	显示一个文本区域用于输入密码
h:outputLabel	显示一个标签
h:outputLink	显示一个超文本链接
h:selectOneMenu	显示一个组合框用于选择一项
h:selectOneRadio	显示一组单选按钮
h:selectManyCheckbox	显示复选框
h:selectOneListbox	显示一个线性表用于选择一项
h:selectManyListbox	显示一个线性表用于选择多项
f:selectItem	在 h:selectOneMenu、h:selectOneRadio 或者 h:selectManyListbox 中选择一项
h:message	显示一条消息用于验证输入
h:dataTable	显示一个数据表格
h:column	在一个数据表格中指定一列
h:graphicImage	显示一个图像

代码清单 33-4 是一个使用这些元素的部分来显示一个学生注册表格的示例，如图 33-11 所示。

The screenshot shows a web browser window titled "Student Registration Form - Mozilla Firefox". The address bar shows a local host path. The form itself is titled "Student Registration Form" and contains the following elements:

- Input fields for "Last Name", "First Name", and "MI".
- Radio buttons for "Gender" with options "Male" and "Female".
- Dropdown menus for "Major" (with "Computer Science" selected) and "Minor" (with "Mathematics" selected).
- Checkboxes for "Hobby" with options "Tennis", "Golf", and "Ping Pong".
- A text area for "Remarks".
- A "Register" button at the bottom.

图 33-11 使用 JSF 元素显示一个学生注册表单

程序清单 33-4 StudentRegistrationForm.xhtml

```

1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5        xmlns:h="http://xmlns.jcp.org/jsf/html"
6        xmlns:f="http://xmlns.jcp.org/jsf/core">
7    <h:head>
8        <title>Student Registration Form</title>
9    </h:head>
10   <h:body>
11       <h:form>
12           <!-- Use h:graphicImage -->
13           <h3>Student Registration Form
14               <h:graphicImage name="usIcon.gif" library="image"/>
15           </h3>
16
17           <!-- Use h:panelGrid -->
18           <h:panelGrid columns="6" style="color:green">
19               <h:outputLabel value="Last Name"/>
20               <h:inputText id="lastNameInputText" />
21               <h:outputLabel value="First Name" />
22               <h:inputText id="firstNameInputText" />
23               <h:outputLabel value="MI" />
24               <h:inputText id="miInputText" size="1" />
25           </h:panelGrid>
26
27           <!-- Use radio buttons -->
28           <h:panelGrid columns="2">
29               <h:outputLabel>Gender </h:outputLabel>
30               <h:selectOneRadio id="genderSelectOneRadio">
31                   <f:selectItem itemValue="Male"
32                       itemLabel="Male"/>
33                   <f:selectItem itemValue="Female"
34                       itemLabel="Female"/>
35               </h:selectOneRadio>
36           </h:panelGrid>
37
38           <!-- Use combo box and list -->
39           <h:panelGrid columns="4">
40               <h:outputLabel value="Major" />
41               <h:selectOneMenu id="majorSelectOneMenu">
42                   <f:selectItem itemValue="Computer Science"/>
43                   <f:selectItem itemValue="Mathematics"/>
44               </h:selectOneMenu>

```



```

45     <h:outputLabel value="Minor" />
46     <h:selectManyListbox id="minorSelectManyListbox">
47         <f:selectItem itemValue="Computer Science"/>
48         <f:selectItem itemValue="Mathematics"/>
49         <f:selectItem itemValue="English"/>
50     </h:selectManyListbox>
51 </h:panelGrid>
52
53 <!-- Use check boxes -->
54 <h:panelGrid columns="4">
55     <h:outputLabel value="Hobby:" />
56     <h:selectManyCheckbox id="hobbySelectManyCheckbox">
57         <f:selectItem itemValue="Tennis"/>
58         <f:selectItem itemValue="Golf"/>
59         <f:selectItem itemValue="Ping Pong"/>
60     </h:selectManyCheckbox>
61 </h:panelGrid>
62
63 <!-- Use text area -->
64 <h:panelGrid columns="1">
65     <h:outputLabel>Remarks:</h:outputLabel>
66     <h:inputTextarea id="remarksInputTextarea"
67         style="width:400px; height:50px;" />
68 </h:panelGrid>
69
70 <!-- Use command button -->
71 <h:commandButton value="Register" />
72 </h:form>
73 </h:body>
74 </html>

```

前缀 f 的标签位于 JSF 核心标签库中，第 6 行

```
xmlns:f="http://xmlns.jcp.org/jsf/core">
```

为这些标签给出库的位置。

h:graphicImage 标签显示文件 usIcon.gif 中的图像（第 14 行）。文件位于 /resources/image 文件夹中。在 JSF2.2 中，所有的资源（图像文件、声音文件、CCS 文件）都应该放在 Web Pages 结点下面的资源文件夹中。可以如下创建这些文件夹：

步骤 1：在项目面板中右击 Web Pages 结点，显示一个上下文菜单，然后选择 New → Folder 来显示 New Folder 对话框。（如果 Folder 不在上下文菜单中，选择 Other 来定位。）

步骤 2：输入 resources 作为 Folder Name（文件夹名字），然后点击 Finish 来创建 resources 文件夹，如图 33-12 所示。

步骤 3：在项目面板中右击 resources 结点来在 resources 目录下创建图像目录。现在可以将 usIcon.gif 放在图像目录下了。

JSF 提供了 h:panelGrid 和 h:panelGroup 元素来包含和放置子元素。h:panelGrid 将元素放置在一个网格中，类似于 JavaFX 中的 GridPane。h:panelGroup 放置元素类似于 JavaFX 中的 FlowPane。第 18 ~ 25 行放置 6 个元素（标签和输入文本）在一个 h:panelGrid 中。columns 属性指定网格中的每行有 6 列。元素以它们在 facelet 中出现

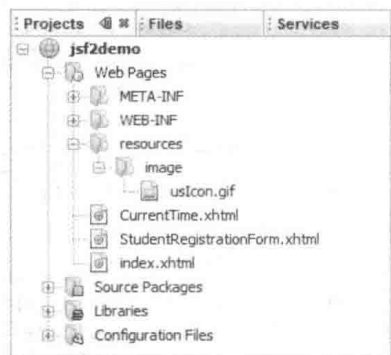


图 33-12 resources 文件夹被创建

的顺序从左到右放置在一行中。当一行满了的时候,新的一行被创建来放置元素。在本例中我们使用了 `h:panelGrid`。你可以将其替换为 `h:panelGroup` 来看看元素将被如何安排。

可以在 JSF 的 html 标签中使用 `style` 属性来为该元素和它的子元素指定 CSS 风格。第 18 行的 `style` 属性为该 `h:panelGrid` 元素中的所有元素指定颜色为绿。

`h:outputLabel` 元素用于显示一个标签 (第 19 行)。`value` 属性给定标签的文本。

`h:inputText` 元素用于显示一个文本输入框,让用户输入一个文本 (第 20 行)。`id` 属性对于其他元素或者服务器程序引用该元素非常有用。

`h:selectOneRadio` 元素用于显示一组单选按钮 (第 30 行)。每个单选按钮使用一个 `f:selectItem` 元素定义 (第 31 ~ 34 行)。

`h:selectOneMenu` 元素用于显示一个组合框 (第 41 行)。该组合框中的每一项使用一个 `f:selectItem` 元素定义 (第 42 行和 43 行)。

`h:selectManyListBox` 元素用于显示一个线性表,让用户可以在一个线性表中选择多项 (第 46 行)。线性表的每一项使用一个 `f:selectItem` 元素定义 (第 47 ~ 49 行)。

`h:selectManyCheckBox` 元素用于显示一组复选框 (第 56 行)。复选框中的每一项使用一个 `f:selectItem` 元素定义 (第 57 ~ 59 行)。

`h:selectTextArea` 元素用于显示一个可以多行输入的文本区域 (第 66 行)。`style` 属性用于指定该文本区域的宽度和高度 (第 67 行)。

`h:commandButton` 元素用于显示一个按钮 (第 71 行)。当按钮被点击,一个动作被执行。默认的动作是从服务器请求同一个页面。下一节中给出如何处理表单。

✓ 复习题

33.9 前缀为 `h` 以及 `f` 的 JSF 标签的名称空间是什么?

33.10 描述以下标签的作用?

```
h:form, h:panelGroup, h:panelGrid, h:inputText, h:outputText,
h:inputTextArea, h:inputSecret, h:outputLabel, h:outputLink,
h:selectOneMenu, h:selectOneRadio, h:selectManyCheckbox,
h:selectOneListbox, h:selectManyListbox, h:selectItem, h:message,
h:dataTable, h:column, h:graphicImage
```

33.4 处理表单

🔑 **要点提示:** 对于 Web 编程而言,处理表单是一个常见的任务。JSF 提供了处理表单的工具。

前面小节介绍了如何使用常用的 JSF 元素来显示一个表单。本节演示如何获取以及处理输入。

要从表单获取输入,可以简单地将每个输入元素和受管 bean 的属性进行绑定。现在我们定义一个名为 `registration` 的受管 bean,如代码清单 33-5 所示。

程序清单 33-5 RegistrationJSFBean.java

```
1 package jsf2demo;
2
3 import javax.enterprise.context.RequestScoped;
4 import javax.inject.Named;
5
6 @Named(value = "registration")
7 @RequestScoped
```

```
8 public class RegistrationJSFBean {
9     private String lastName;
10    private String firstName;
11    private String mi;
12    private String gender;
13    private String major;
14    private String[] minor;
15    private String[] hobby;
16    private String remarks;
17
18    public RegistrationJSFBean() {
19    }
20
21    public String getLastName() {
22        return lastName;
23    }
24
25    public void setLastName(String lastName) {
26        this.lastName = lastName;
27    }
28
29    public String getFirstName() {
30        return firstName;
31    }
32
33    public void setFirstName(String firstName) {
34        this.firstName = firstName;
35    }
36
37    public String getMi() {
38        return mi;
39    }
40
41    public void setMi(String mi) {
42        this.mi = mi;
43    }
44
45    public String getGender() {
46        return gender;
47    }
48
49    public void setGender(String gender) {
50        this.gender = gender;
51    }
52
53    public String getMajor() {
54        return major;
55    }
56
57    public void setMajor(String major) {
58        this.major = major;
59    }
60
61    public String[] getMinor() {
62        return minor;
63    }
64
65    public void setMinor(String[] minor) {
66        this.minor = minor;
67    }
68
69    public String[] getHobby() {
70        return hobby;
71    }
72
73    public void setHobby(String[] hobby) {
```

```

74     this.hobby = hobby;
75 }
76
77 public String getRemarks() {
78     return remarks;
79 }
80
81 public void setRemarks(String remarks) {
82     this.remarks = remarks;
83 }
84
85 public String getResponse() {
86     if (lastName == null)
87         return ""; // Request has not been made
88     else {
89         String allMinor = "";
90         for (String s: minor) {
91             allMinor += s + " ";
92         }
93
94         String allHobby = "";
95         for (String s: hobby) {
96             allHobby += s + " ";
97         }
98
99         return "<p style='color:red'>You entered <br />" +
100             "Last Name: " + lastName + "<br />" +
101             "First Name: " + firstName + "<br />" +
102             "MI: " + mi + "<br />" +
103             "Gender: " + gender + "<br />" +
104             "Major: " + major + "<br />" +
105             "Minor: " + allMinor + "<br />" +
106             "Hobby: " + allHobby + "<br />" +
107             "Remarks: " + remarks + "</p>";
108     }
109 }
110 }

```

RegistrationJSFBean 类是一个受管 bean，定义了属性 lastName、firstName、mi、gender、major、minor 以及 remarks。这些属性将绑定到 JSF 注册表单中的元素上。

现在注册表单可以修改如程序清单 33-6 所示。图 33-13 显示了当用户点击 Register 按钮时，新的 JSF 页面显示用户输入。

程序清单 33-6 ProcessStudentRegistrationForm.xhtml

```

1 <?xml version='1.0' encoding='UTF-8' ?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml">
5     xmlns:h="http://xmlns.jcp.org/jsf/html"
6     xmlns:f="http://xmlns.jcp.org/jsf/core">
7 <h:head>
8     <title>Student Registration Form</title>
9 </h:head>
10 <h:body>
11     <h:form>
12         <!-- Use h:graphicImage -->
13         <h3>Student Registration Form
14             <h:graphicImage name="usIcon.gif" library="image"/>
15         </h3>
16
17         <!-- Use h:panelGrid -->
18         <h:panelGrid columns="6" style="color:green">
19             <h:outputLabel value="Last Name"/>

```

```

19     <h:outputLabel value="Last Name"/>
20     <h:inputText id="lastNameInputText"
21         value="#{registration.lastName}"/>
22     <h:outputLabel value="First Name" />
23     <h:inputText id="firstNameInputText"
24         value="#{registration.firstName}"/>
25     <h:outputLabel value="MI" />
26     <h:inputText id="miInputText" size="1"
27         value="#{registration.mi}"/>
28 </h:panelGrid>
29
30 <!-- Use radio buttons -->
31 <h:panelGrid columns="2">
32     <h:outputLabel>Gender </h:outputLabel>
33     <h:selectOneRadio id="genderSelectOneRadio"
34         value="#{registration.gender}">
35         <f:selectItem itemValue="Male"
36             itemLabel="Male"/>
37         <f:selectItem itemValue="Female"
38             itemLabel="Female"/>
39     </h:selectOneRadio>
40 </h:panelGrid>
41
42 <!-- Use combo box and list -->
43 <h:panelGrid columns="4">
44     <h:outputLabel value="Major" />
45     <h:selectOneMenu id="majorSelectOneMenu"
46         value="#{registration.major}">
47         <f:selectItem itemValue="Computer Science"/>
48         <f:selectItem itemValue="Mathematics"/>
49     </h:selectOneMenu>
50     <h:outputLabel value="Minor" />
51     <h:selectManyListbox id="minorSelectManyListbox"
52         value="#{registration.minor}">
53         <f:selectItem itemValue="Computer Science"/>
54         <f:selectItem itemValue="Mathematics"/>
55         <f:selectItem itemValue="English"/>
56     </h:selectManyListbox>
57 </h:panelGrid>
58
59 <!-- Use check boxes -->
60 <h:panelGrid columns="4">
61     <h:outputLabel value="Hobby: " />
62     <h:selectManyCheckbox id="hobbySelectManyCheckbox"
63         value="#{registration.hobby}">
64         <f:selectItem itemValue="Tennis"/>
65         <f:selectItem itemValue="Golf"/>
66         <f:selectItem itemValue="Ping Pong"/>
67     </h:selectManyCheckbox>
68 </h:panelGrid>
69
70 <!-- Use text area -->
71 <h:panelGrid columns="1">
72     <h:outputLabel>Remarks:</h:outputLabel>
73     <h:inputTextarea id="remarksInputTextarea"
74         style="width:400px; height:50px;"
75         value="#{registration.remarks}"/>
76 </h:panelGrid>
77
78 <!-- Use command button -->
79 <h:commandButton value="Register" />
80 <br />
81 <h:outputText escape="false" style="color:red"
82     value="#{registration.response}" />
83 </h:form>

```

```
84 </h:body>
85 </html>
```

该清单中的新的 JSF 表单将用于姓、名以及中间名字的 `h:inputText` 元素和受管 bean 中的属性 `lastName`、`firstName` 以及 `mi` 进行了绑定 (第 21、24、27 行)。当点击 Register 按钮时, 页面发送给服务器, 服务器调用 `setter` 方法从而设置受管 bean 中的属性。

`h:selectOneRadio` 元素绑定到 `gender` 属性 (第 34 行)。每个单选按钮有一个 `itemValue`。当页面发送给服务器, 被选择的单选按钮的 `itemValue` 将赋给 bean 中的 `gender` 属性。

`h:selectOneMenu` 元素绑定到 `major` 属性 (第 46 行)。当页面发送给服务器, 被选择的条目以字符串返回并赋给 `major` 属性。

`h:selectManyListbox` 元素绑定到 `minor` 属性 (第 52 行)。当页面发送给服务器, 被选择的条目以字符串数组返回并赋给 `minor` 属性。

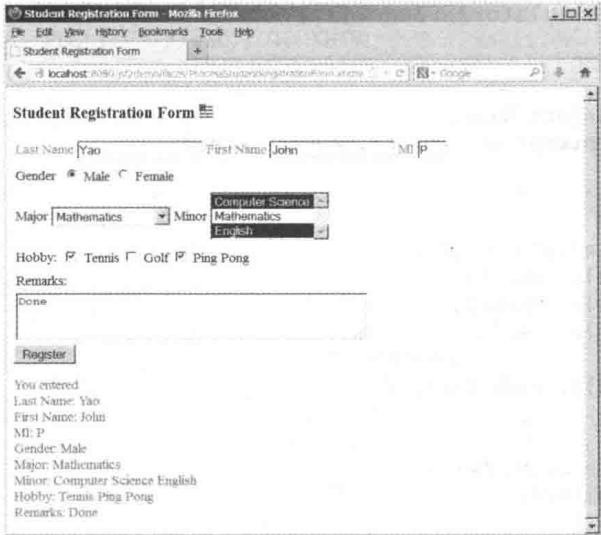


图 33-13 点击 Register 按钮后, 用户的输入被收集并且显示

`h:selectManyCheckbox` 元素绑定到 `hobby` 属性 (第 63 行)。当页面发送给服务器, 被勾选的项以 `itemValues` 数组返回并赋给 `hobby` 属性。

`h:selectTextarea` 元素绑定到 `remarks` 属性 (第 75 行)。当页面发送给服务器, 文本区域中的内容以字符串返回并赋给 `remarks` 属性。

`h:outputText` 元素绑定到 `response` 属性 (第 82 行)。这是 bean 中的一个只读属性。如果 `lastName` 为 `null` (代码清单 33-5 中的第 86 ~ 87 行), 则属性值为 `""`。当页面返回给客户端, `response` 属性值显示在输出文本元素中 (第 82 行)。

`h:outputText` 元素的 `escape` 属性设置为 `false` (第 81 行), 从而使得内容可以显示为 HTML 格式。默认的, `escape` 属性值为 `true`, 这表明内容被当做一种常规的文本。

✓ 复习题

- 33.11 `h:outputText` 标签中的 `escape` 属性的作用是什么?
- 33.12 是否 JSF 中的每个 GUI 组件标签都有一个 `style` 属性?

33.5 示例学习: 计算器

🔑 要点提示: 本节给出一个使用 GUI 元素和表单处理的示例学习。

本节使用 JSF 来开发一个计算器，执行加法、减法、乘法以及除法，如图 33-14 所示。

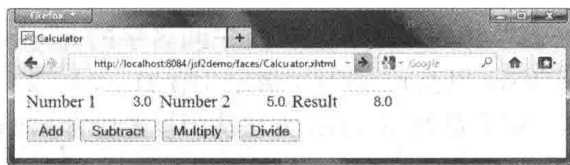


图 33-14 该 JSF 应用可以执行加法、减法、乘法以及除法运算

下面是开发该项目的步骤：

步骤 1：创建一个名为 calculator 的新的受管 bean，具有 request（访问）范围，如程序清单 33-7 所示。

步骤 2：创建一个 JSF facelet，如程序清单 33-8 所示。

程序清单 33-7 CalculatorJSFBean.java

```
1 package jsf2demo;
2
3 import javax.inject.Named;
4 import javax.enterprise.context.RequestScoped;
5
6 @Named(value = "calculator")
7 @RequestScoped
8 public class CalculatorJSFBean {
9     private Double number1;
10    private Double number2;
11    private Double result;
12
13    public CalculatorJSFBean() {
14    }
15
16    public Double getNumber1() {
17        return number1;
18    }
19
20    public Double getNumber2() {
21        return number2;
22    }
23
24    public Double getResult() {
25        return result;
26    }
27
28    public void setNumber1(Double number1) {
29        this.number1 = number1;
30    }
31
32    public void setNumber2(Double number2) {
33        this.number2 = number2;
34    }
35
36    public void setResult(Double result) {
37        this.result = result;
38    }
39
40    public void add() {
41        result = number1 + number2;
42    }
43
44    public void subtract() {
45        result = number1 - number2;
```



```

46     }
47
48     public void divide() {
49         result = number1 / number2;
50     }
51
52     public void multiply() {
53         result = number1 * number2;
54     }
55 }

```

该受管 bean 具有三个属性 `number1`、`number2` 和 `result` (第 9 ~ 38 行)。方法 `add()`、`subtract()`、`divide()` 以及 `multiply()` 将 `number1` 和 `number2` 进行相加、相减、相乘, 并将结果赋值给 `result` (第 40 ~ 54 行)。

程序清单 33-8 Calculator.xhtml

```

1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5    xmlns:h="http://xmlns.jcp.org/jsf/html">
6    <h:head>
7        <title>Calculator</title>
8    </h:head>
9    <h:body>
10        <h:form>
11            <h:panelGrid columns="6">
12                <h:outputLabel value="Number 1"/>
13                <h:inputText id="number1InputText" size="4"
14                    style="text-align: right"
15                    value="#{calculator.number1}"/>
16                <h:outputLabel value="Number 2" />
17                <h:inputText id="number2InputText" size="4"
18                    style="text-align: right"
19                    value="#{calculator.number2}"/>
20                <h:outputLabel value="Result" />
21                <h:inputText id="resultInputText" size="4"
22                    style="text-align: right"
23                    value="#{calculator.result}"/>
24            </h:panelGrid>
25
26            <h:panelGrid columns="4">
27                <h:commandButton value="Add"
28                    action="#{calculator.add}"/>
29                <h:commandButton value="Subtract"
30                    action="#{calculator.subtract}"/>
31                <h:commandButton value="Multiply"
32                    action="#{calculator.multiply}"/>
33                <h:commandButton value="Divide"
34                    action="#{calculator.divide}"/>
35            </h:panelGrid>
36        </h:form>
37    </h:body>
38 </html>

```

3 个文本输入组件以及它们的标签放置在一个网格面板中 (第 11 ~ 24 行)。4 个按钮组件放置在网格面板中 (第 26 ~ 35 行)。

bean 属性 `number1` 绑定到 Number 1 的文本输入 (第 15 行)。CSS 风格 `text-align:right` (第 14 行) 指定文本在输入框中右对齐。

Add 按钮的 `action` 属性被设在计算 bean 的 `add` 方法上 (第 28 行)。当 Add 按钮被点击

时, bean 中的 add 方法被调用, 将 number1 和 number2 相加并将结果赋值给 result。由于 result 属性绑定到 Result 输入本文框中 (第 23 行), 现在新的结果显示在文本输入域中。

33.6 会话跟踪

要点提示: 可以创建一个具有应用程序范围、会话范围、视图范围或者请求范围的受管 bean。

JSF 通过具有应用程序范围、会话范围、视图范围或者请求范围的 JavaBean 支持会话跟踪。scope (范围) 是指一个 bean 的生命周期。request-scoped (请求范围) bean 在一个 HTTP 请求范围内存在。当请求被处理, bean 不再存在。view-scoped (视图范围) bean 在你停留在同一个 JSF 页面期间一直存在。session-scoped (会话范围) bean 在一个客户端和服务端之间的整个 Web 会话范围内存在。只要 Web 应用运行, application-scoped (应用程序范围) bean 都是存在的。本质上, 一个请求范围内的 bean 对于一个请求创建一次; 一个视图范围内的 bean 对于视图创建一次; 一个会话范围的 bean 对于整个会话创建一次; 一个应用程序范围的 bean 对于整个应用创建一次。

考虑下面提示用户猜测数字的示例。当页面启动, 程序随机地生成一个 0 和 99 之间的数字。该数字保存在一个 bean 中。当用户输入一个猜测的数字, 程序将该猜测的数字和 bean 中的随机数比较, 然后告诉用户这个猜测的数字是否偏高、偏低或者猜中了, 如图 33-15 所示。



图 33-15 用户输入一个猜测的数字, 程序显示结果

这里是开发该项目的步骤:

步骤 1: 创建一个名为 guessNumber 的新的具有视图范围的受管 bean。如程序清单 33-9

所示。

步骤 2: 创建一个程序清单 33-10 中的 JSF facelet。

程序清单 33-9 GuessNumberJSFBean.java

```

1 package jsf2demo;
2
3 import javax.inject.Named;
4 import javax.faces.view.ViewScoped;
5
6 @Named(value = "guessNumber")
7 @ViewScoped
8 public class GuessNumberJSFBean {
9     private int number;
10    private String guessString;
11
12    public GuessNumberJSFBean() {
13        number = (int)(Math.random() * 100);
14    }
15
16    public String getGuessString() {
17        return guessString;
18    }
19
20    public void setGuessString(String guessString) {
21        this.guessString = guessString;
22    }
23
24    public String getResponse() {
25        if (guessString == null)
26            return ""; // No user input yet
27
28        int guess = Integer.parseInt(guessString);
29        if (guess < number)
30            return "Too low";
31        else if (guess == number)
32            return "You got it";
33        else
34            return "Too high";
35    }
36 }

```

受管 bean 使用 `@ViewScope` 标注 (第 7 行) 来为 bean 设置一个视图范围。对该项目来说, 视图范围是最合适的。只要视图不改变, bean 就存在。当页面第一次显示的时候, bean 被创建。当 bean 被创建, 一个 0 到 99 之间的随机数被赋给 `number` (第 13 行)。只要 bean 在同一个视图中存在, 该数字不会变化。

`getResponse` 方法将用户输入的 `guessString` 转换为一个整数 (第 28 行), 并确定该猜测的数字是偏低 (第 30 行)、偏高 (第 34 行) 或者正好 (第 32 行)。

程序清单 33-10 GuessNumber.xhtml

```

1 <?xml version='1.0' encoding='UTF-8' ?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml"
5       xmlns:h="http://xmlns.jcp.org/jsf/html">
6   <h:head>
7       <title>Guess a number</title>
8   </h:head>
9   <h:body>
10      <h:form>
11          <h:outputLabel value="Enter you guess: "/>

```

```
12      <h:inputText style="text-align: right; width: 50px"
13                  id="guessInputText"
14                  value="#{guessNumber.guessString}"/>
15      <h:commandButton style="margin-left: 60px" value="Guess" />
16      <br />
17      <h:outputText style="color: red"
18                   value="#{guessNumber.response}" />
19  </h:form>
20 </h:body>
21 </html>
```

bean 属性 guessString 绑定到文本输入域中（第 14 行）。CSS 样式 text-align:right(第 13 行) 指定文本在输入框中右对齐。

CSS 样式 margin-left:60px（第 15 行）指定命令按钮距离左边边距为 60 像素。

bean 的属性 response 绑定到文本输出域上（第 18 行）。CSS 样式 color:red（第 17 行）指定文本在输出框中显示为红色。

项目使用了视图范围。如果该范围改为请求范围，将会如何？每次页面被刷新，JSF 将会创建一个具有新的随机数的新的 bean。如果范围改为会话范围，将会如何？只要浏览器没有关闭，bean 将存在。如果范围改为应用范围将会如何？当服务器启动应用后，bean 只会被创建一次。因此，所有的用户将使用同样一个随机数字。

✓ 复习题

- 33.13 什么是范围？JSF 中可用的范围是什么？解释请求范围、视图范围、会话范围以及应用范围。如何在受管 bean 中设置请求范围、视图范围、会话范围以及应用范围？
- 33.14 如果程序清单 33-9 中的 bean 范围修改为请求范围，会如何？
- 33.15 如果程序清单 33-9 中的 bean 范围修改为会话范围，会如何？
- 33.16 如果程序清单 33-9 中的 bean 范围修改为应用范围，会如何？

33.7 验证输入

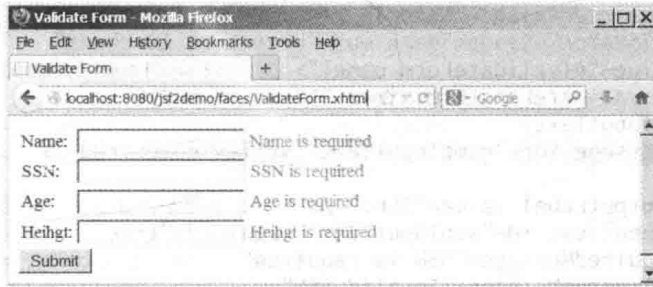
🔑 要点提示：JSF 提供了验证用户输入的工具。

在前面的 GuessNumber 页面中，如果在点击 Guess 按钮之前在输入框中输入了一个非整数，将会出错。解决这个问题的一种办法是处理任何事件前检查文本域。但是一个更好的方式是使用验证器。可以使用 JSF Core Tag Library（JSF 核心标签库）中的标准验证器，或者创建自定义的验证器。表 33-2 列出了一些 JSF 输入验证器标签。

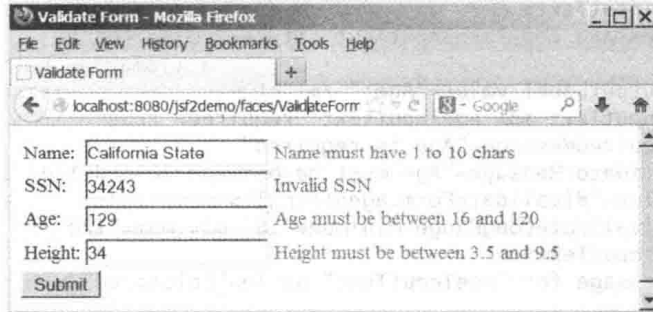
表 33-2 JSF 输入验证器标签

JSF 标签	描述
f:validateLength	验证输入的长度
f:validateDoubleRange	验证输入的数字是否在双精度值的可接受范围内
f:validateLongRange	验证输入的数字是否在长整数值的可接受范围内
f:validateRequired	验证一个字段是否非空
f:validateRegex	验证输入是否符合一个正则表达式
f:validateBean	调用 bean 中的一个自定义方法来执行一个自定义的验证

考虑下面显示一个表单来收集用户输入的示例，如图 33-16 所示。表单中所有的文本域必须被填充。如果没有，则显示一个错误消息。SSN 必须具有正确的格式。如果不具有，显示一个错误。如果所有的输入都是正确的，点击 Submit 在一个输出文本中显示结果，如图 33-17 所示。



a) 如果要求有输入但是为空，则显示要求输入的信息



b) 如果输入不正确，则显示错误信息

图 33-16 输入域被验证

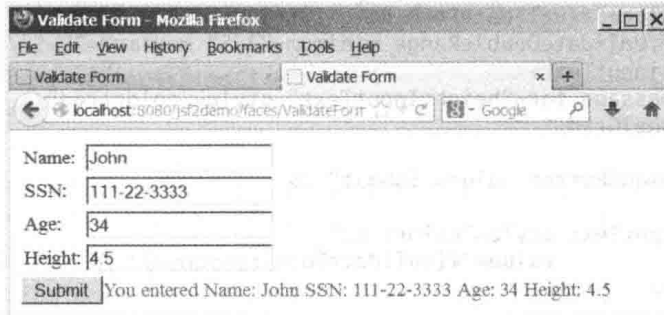


图 33-17 正确输入的值被显示

这里是创建该项目的步骤：

步骤 1：创建程序清单 33-11 中的新页面。

步骤 2：创建一个名为 `validateForm` 的新的受管 bean，如程序清单 33-12 所示。

程序清单 33-11 ValidateForm.xhtml

```

1 <?xml version='1.0' encoding='UTF-8' ?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml"
5       xmlns:h="http://xmlns.jcp.org/jsf/html"
6       xmlns:f="http://xmlns.jcp.org/jsf/core">
7   <h:head>
8     <title>Validate Form</title>
9   </h:head>
10  <h:body>
11    <h:form>
12      <h:panelGrid columns="3">
13        <h:outputLabel value="Name:"/>
14        <h:inputText id="nameInputText" required="true"

```

```

15         requiredMessage="Name is required"
16         validatorMessage="Name must have 1 to 10 chars"
17         value="#{validateForm.name}">
18         <f:validateLength minimum="1" maximum="10" />
19     </h:inputText>
20     <h:message for="nameInputText" style="color:red"/>
21
22     <h:outputLabel value="SSN:" />
23     <h:inputText id="ssnInputText" required="true"
24         requiredMessage="SSN is required"
25         validatorMessage="Invalid SSN"
26         value="#{validateForm.ssn}">
27         <f:validateRegex pattern="\d{3}-\d{2}-\d{4}" />
28     </h:inputText>
29     <h:message for="ssnInputText" style="color:red"/>
30
31     <h:outputLabel value="Age:" />
32     <h:inputText id="ageInputText" required="true"
33         requiredMessage="Age is required"
34         validatorMessage="Age must be between 16 and 120"
35         value="#{validateForm.ageString}">
36         <f:validateLongRange minimum="16" maximum="120"/>
37     </h:inputText>
38     <h:message for="ageInputText" style="color:red"/>
39
40     <h:outputLabel value="Height:" />
41     <h:inputText id="heightInputText" required="true"
42         requiredMessage="Height is required"
43         validatorMessage="Height must be between 3.5 and 9.5"
44         value="#{validateForm.heightString}">
45         <f:validateDoubleRange minimum="3.5" maximum="9.5"/>
46     </h:inputText>
47     <h:message for="heightInputText" style="color:red"/>
48 </h:panelGrid>
49
50 <h:commandButton value="Submit" />
51
52 <h:outputText style="color:red"
53     value="#{validateForm.response}" />
54 </h:form>
55 </h:body>
56 </html>

```

对于每一个输入的文本域，设置其 `required` 属性为 `true` (第 14、23、32、41 行)，表示该域要求有一个输入值。当一个要求的输入域为空，则显示 `requiredMessage` (第 15、24、33、42 行)。

`validatorMessage` 属性指定在输入域无效时显示的信息 (第 16 行)。`f:validateLength` 标签指定输入的最小和最大长度 (第 18 行)。JSF 将确定输入长度是否有效。

`h:message` 元素在输入无效时显示 `validatorMessage`。元素的 `for` 属性指定将显示的消息所针对的元素 `id` (第 20 行)。

`f:validateRegex` 标签指定验证输入的正则表达式 (第 27 行)。关于正则表达式的信息，参见附录 H。

`f:validateLongRange` 标签使用 `minimum` 和 `maximum` 属性为一个整数输入指定范围 (第 36 行)。在该项目中，一个有效的年龄值位于 16 到 120 之间。

`f:validateDoubleRange` 标签使用 `minimum` 和 `maximum` 属性为一个双精度值输入指定范围 (第 45 行)。在该项目中，一个有效的身高值处于 3.5 到 9.5 之间。

程序清单 33-12 ValidateFormJSFBean.java

```
1 package jsf2demo;
2
3 import javax.enterprise.context.RequestScoped;
4 import javax.inject.Named;
5
6 @Named(value = "validateForm")
7 @RequestScoped
8 public class ValidateFormJSFBean {
9     private String name;
10    private String ssn;
11    private String ageString;
12    private String heightString;
13
14    public String getName() {
15        return name;
16    }
17
18    public void setName(String name) {
19        this.name = name;
20    }
21
22    public String getSsn() {
23        return ssn;
24    }
25
26    public void setSsn(String ssn) {
27        this.ssn = ssn;
28    }
29
30    public String getAgeString() {
31        return ageString;
32    }
33
34    public void setAgeString(String ageString) {
35        this.ageString = ageString;
36    }
37
38    public String getHeightString() {
39        return heightString;
40    }
41
42    public void setHeightString(String heightString) {
43        this.heightString = heightString;
44    }
45
46    public String getResponse() {
47        if (name == null || ssn == null || ageString == null
48            || heightString == null) {
49            return "";
50        }
51        else {
52            return "You entered " +
53                " Name: " + name +
54                " SSN: " + ssn +
55                " Age: " + ageString +
56                " Height: " + heightString;
57        }
58    }
59 }
```

如果一个输入是无效的，它的值不会发送给 bean。因此，只有当所有输入都是正确的，getResponse() 方法将返回所有的输入值（第 46 ~ 58 行）。

复习题

- 33.17 编写一个标签，可以验证一个文本输入，使得最小长度为 2，最大长度为 12。
- 33.18 编写一个标签，可以使用正则表达式验证一个作为文本输入的 SSN。
- 33.19 编写一个标签，可以验证一个作为文本输入的双精度值，使得最小值为 4.5，最大值为 19.9。
- 33.20 编写一个标签，可以验证一个作为文本输入的整数值，使得最小值为 4，最大值为 20。
- 33.21 编写一个标签，使得必须输入一个文本。

33.8 将数据库与 facelet 绑定

要点提示：可以在一个 JSF 应用中绑定数据库。

通常，需要从一个 Web 页面中访问数据库。本节给出使用数据库构建 Web 应用的例子。

考虑下面让用户选择一门课的示例，如图 33-18 所示。当在一个组合框中选择好一门课后，注册该课程的学生在表格中显示，如图 33-19 所示。在该示例中，Course 表中的所有课程名称绑定到组合框中，而查询注册该课程的学生查询结果绑定到表格中。

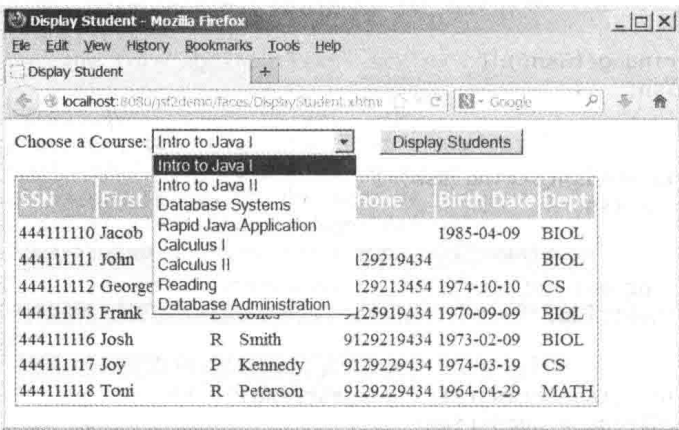


图 33-18 你需要选择一门课，然后会显示注册该课程中的学生

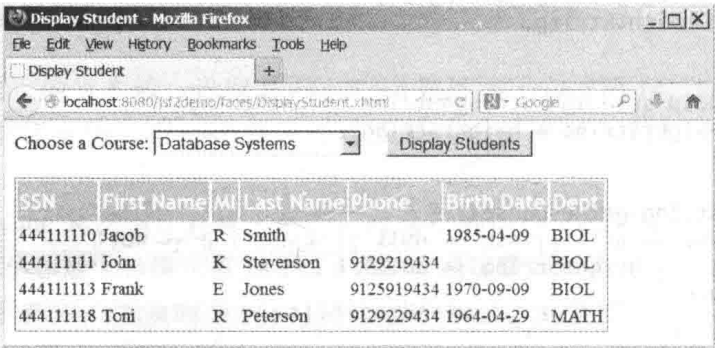


图 33-19 表格显示注册该课程的学生

下面是创建该项目的步骤。

步骤 1：创建一个具有应用范围名为 `courseName` 的受管 bean，如程序清单 33-13 所示。

步骤 2：创建一个程序清单 33-14 中的 JSF 页面。

步骤 3：如下为格式化表格创建一个层叠样式表：

步骤 3.1：右击 `resources` 结点，选择 `New → Others` 以显示 `New File` 对话框，如图 33-20 所示。

步骤 3.2：在 Categories 部分选择 Other，在 File Types 部分选择 Cascading Style Sheet，显示 New Cascading Style Sheet 对话框，如图 33-21 所示。

步骤 3.3：输入 tablestyle 作为 File Name，点击 Finish 从而在资源结点下面创建 tablestyle.css。

步骤 3.4：如程序清单 33-15，定义 CSS 样式。

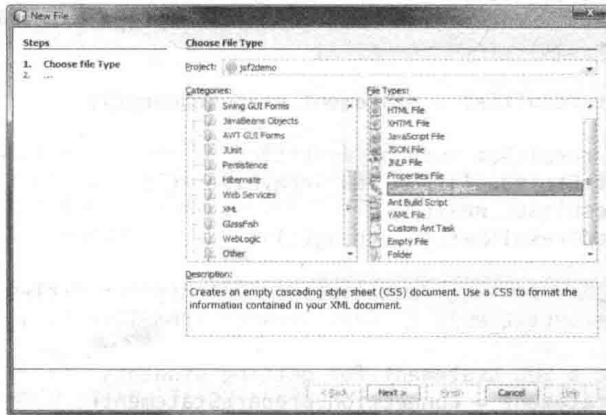


图 33-20 可以在 NetBeans 中为 Web 项目创建 CSS 文件

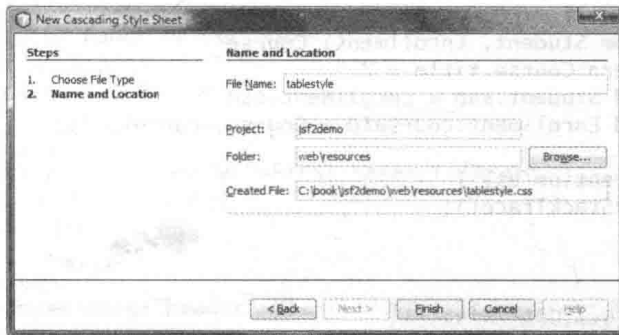


图 33-21 New Cascading Style Sheet 对话框创建一个新的样式表文件

程序清单 33-13 CourseNameJSFBean.java

```

1 package jsf2demo;
2
3 import java.sql.*;
4 import java.util.ArrayList;
5 import javax.enterprise.context.ApplicationScoped;
6 import javax.inject.Named;
7
8 @Named(value = "courseName")
9 @ApplicationScoped
10 public class CourseNameJSFBean {
11     private PreparedStatement studentStatement = null;
12     private String choice; // Selected course
13     private String[] titles; // Course titles
14
15     /** Creates a new instance of CourseName */
16     public CourseNameJSFBean() {
17         initializeJdbc();
18     }
19
20     /** Initialize database connection */
21     private void initializeJdbc() {

```

```

22 try {
23     Class.forName("com.mysql.jdbc.Driver");
24     System.out.println("Driver loaded");
25
26     // Connect to the sample database
27     Connection connection = DriverManager.getConnection(
28         "jdbc:mysql://localhost/javabook", "scott", "tiger");
29
30     // Get course titles
31     PreparedStatement statement = connection.prepareStatement(
32         "select title from course");
33
34     ResultSet resultSet = statement.executeQuery();
35
36     // Store resultSet into array titles
37     ArrayList<String> list = new ArrayList<>();
38     while (resultSet.next()) {
39         list.add(resultSet.getString(1));
40     }
41     titles = new String[list.size()]; // Array for titles
42     list.toArray(titles); // Copy strings from list to array
43
44     // Define a SQL statement for getting students
45     studentStatement = connection.prepareStatement(
46         "select Student.ssn, "
47         + "Student.firstName, Student.mi, Student.lastName, "
48         + "Student.phone, Student.birthDate, Student.street, "
49         + "Student.zipCode, Student.deptId "
50         + "from Student, Enrollment, Course "
51         + "where Course.title = ? "
52         + "and Student.ssn = Enrollment.ssn "
53         + "and Enrollment.courseId = Course.courseId");
54 }
55 catch (Exception ex) {
56     ex.printStackTrace();
57 }
58 }
59
60 public String[] getTitles() {
61     return titles;
62 }
63
64 public String getChoice() {
65     return choice;
66 }
67
68 public void setChoice(String choice) {
69     this.choice = choice;
70 }
71
72 public ResultSet getStudents() throws SQLException {
73     if (choice == null) {
74         if (titles.length == 0)
75             return null;
76         else
77             studentStatement.setString(1, titles[0]);
78     }
79     else {
80         studentStatement.setString(1, choice); // Set course title
81     }
82
83     // Get students for the specified course
84     return studentStatement.executeQuery();
85 }
86 }

```

程序清单 33-14 DisplayStudent.xhtml

```

1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5    xmlns:h="http://xmlns.jcp.org/jsf/html"
6    xmlns:f="http://xmlns.jcp.org/jsf/core">
7    <h:head>
8      <title>Display Student</title>
9      <h:outputStylesheet name="tablestyle.css" />
10   </h:head>
11   <h:body>
12     <h:form>
13       <h:outputLabel value="Choose a Course: " />
14       <h:selectOneMenu value="#{courseName.choice}"
15         <f:selectItems value="#{courseName.titles}" />
16     </h:selectOneMenu>
17
18     <h:commandButton style="margin-left: 20px"
19       value="Display Students" />
20
21     <br /> <br />
22     <h:dataTable value="#{courseName.students}" var="student"
23       rowClasses="oddTableRow, evenTableRow"
24       headerClass="tableHeader"
25       styleClass="table">
26       <h:column>
27         <f:facet name="header">SSN</f:facet>
28         #{student.ssn}
29       </h:column>
30
31       <h:column>
32         <f:facet name="header">First Name</f:facet>
33         #{student.firstName}
34       </h:column>
35
36       <h:column>
37         <f:facet name="header">MI</f:facet>
38         #{student.mi}
39       </h:column>
40
41       <h:column>
42         <f:facet name="header">Last Name</f:facet>
43         #{student.lastName}
44       </h:column>
45
46       <h:column>
47         <f:facet name="header">Phone</f:facet>
48         #{student.phone}
49       </h:column>
50
51       <h:column>
52         <f:facet name="header">Birth Date</f:facet>
53         #{student.birthDate}
54       </h:column>
55
56       <h:column>
57         <f:facet name="header">Dept</f:facet>
58         #{student.deptId}
59       </h:column>
60     </h:dataTable>
61   </h:form>
62 </h:body>
63 </html>

```

程序清单 33-15 tablestyle.css

```

1  /* Style for table */
2  .tableHeader {
3      font-family:"Trebuchet MS", Arial, Helvetica, sans-serif;
4      border-collapse:collapse;
5      font-size:1.1em;
6      text-align:left;
7      padding-top:5px;
8      padding-bottom:4px;
9      background-color:#A7C942;
10     color:white;
11     border:1px solid #98bf21;
12 }
13
14 .oddTableRow {
15     border:1px solid #98bf21;
16 }
17
18 .evenTableRow {
19     background-color: #eeeeee;
20     font-size:1em;
21
22     padding:3px 7px 2px 7px;
23
24     color:#000000;
25     background-color:#EAF2D3;
26 }
27
28 .table {
29     border:1px solid green;
30 }

```

我们使用第 32 章中创建的同个 MySQL 数据库 javabook。该受管 bean 的范围是 application。从服务器运行该项目时创建该 bean。initializeJdbc 方法为 MySQL 装载 JDBC 驱动程序 (第 23 ~ 24 行), 连接到 MySQL 数据库 (第 27 ~ 28 行), 为获取课程名称创建语句 (第 31 ~ 32 行), 为获取指定课程的学生信息创建语句 (第 45 ~ 53 行)。第 31 ~ 42 行执行获取课程名称的语句, 并将它们存储在数组 titles 中。

getStudents() 方法返回一个包含注册到指定课程的所有学生信息的 ResultSet (第 72 ~ 85 行)。对课程名称的选择设置在语句中, 从而获得指定课程名称的学生 (第 80 行)。如果选择为 null, 则在语句中设置为标题数组中的第一个课程名称 (第 77 行)。如果课程没有名称信息, getStudents() 返回 null (第 75 行)。

提示: 为了在项目中使 MySQL, 需要在 NetBeans 的 Project 面板中的 Libraries 结点中添加 MySQL JDBC 驱动。

第 9 行指定在步骤 3 中创建的样式表 tablestyle.css 在应用到该 XHTML 文件中。rowClasses = "oddTableRow, evenTableRow" 属性指定交替应用 oddTableRow 和 evenTableRow 样式到行上 (第 23 行)。headerClasses = "tableHeader" 属性确定 tableHeader 类用于头部样式 (第 24 行)。styleClasses = "table" 属性指定 table 类作为表中的所有其他元素的样式 (第 25 行)。

第 14 行将 courseName bean 的 choice 属性和组合框绑定。组合框中的选择值和 titles 数组属性绑定 (第 15 行)。

第 22 行使用属性 value = "#{courseName.students}" 将表值和数据库结果集绑定。var = "student" 属性通过 student 和结果集合中的一行关联。第 26 ~ 59 行使用 student.

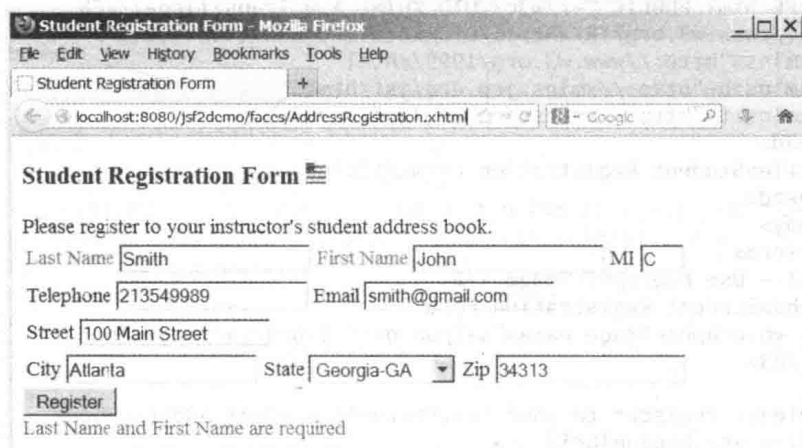
ssn (第 28 行)、student.firstName (第 33 行)、student.mi (第 38 行)、student.lastName (第 33 行)、student.phone (第 48 行)、student.birthDate (第 53 行) 以及 student.deptId (第 58 行) 给定列的值。

样式表文件为表格头部样式定义了样式类 tableHeader (第 2 行), 为表格的奇数行定义了 oddTableRow 类 (第 14 行), 为表格的偶数行定义了 evenTableRow 类 (第 18 行), 以及为其他表格元素定义了 table 类 (第 28 行)。

33.9 打开一个新的 JSF 页面

要点提示: 可以从当前的 JSF 页面打开新的 JSF 页面。

至今为止的所有例子都是项目中只有一个 JSF 页面。假设希望将学生信息注册到数据库中。程序首先显示如图 33-22 的页面来收集学生信息。当用户输入信息并点击 Submit 按钮, 显示一个新的页面来让用户确认输入, 如图 33-23 所示。如果用户点击 Confirm 按钮, 数据保存到数据库中并显示状态页面, 如图 33-24 所示。当用户点击 Go Back 按钮, 返回到第一个页面。



Student Registration Form - Mozilla Firefox

File Edit View History Bookmarks Tools Help

Student Registration Form +

localhost:8080/jsf2demo/faces/AddressRegistration.xhtml | Google

Student Registration Form

Please register to your instructor's student address book.

Last Name First Name MI

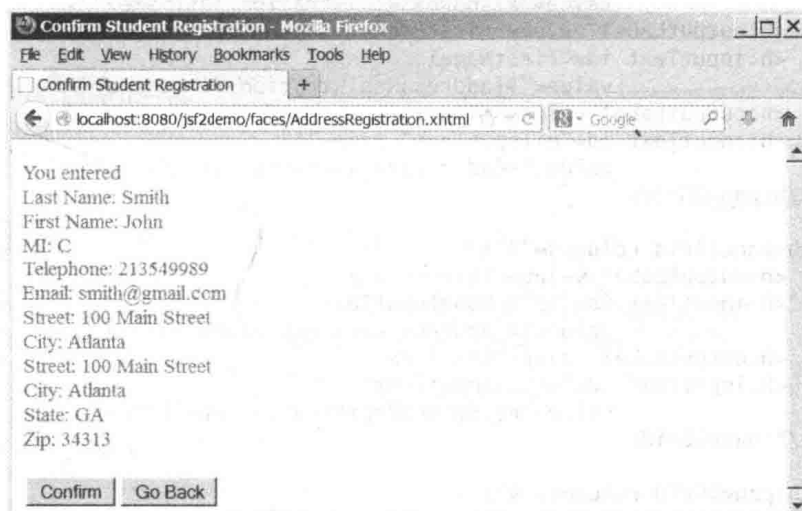
Telephone Email

Street

City State Zip

Last Name and First Name are required

图 33-22 该页面让用户输入



Confirm Student Registration - Mozilla Firefox

File Edit View History Bookmarks Tools Help

Confirm Student Registration +

localhost:8080/jsf2demo/faces/AddressRegistration.xhtml | Google

You entered

Last Name: Smith

First Name: John

MI: C

Telephone: 213549989

Email: smith@gmail.com

Street: 100 Main Street

City: Atlanta

Street: 100 Main Street

City: Atlanta

State: GA

Zip: 34313

图 33-23 该页面让用户确认输入

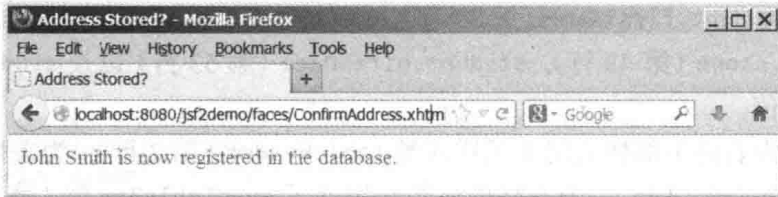


图 33-24 该页面显示用户输入状态

对于该项目，需要创建三个名为 AddressRegistration.xhtml、ConfirmAddress.xhtml、AddressStoredStatus.xhtml 的 JSF 页面，如程序清单 33-16 ~ 程序清单 33-18。项目从 AddressRegistration.xhtml 开始。当点击 Submit 按钮时，如果姓和名不为空，该按钮的动作结果返回 “ConfirmAddress”，这将导致 ConfirmAddress.xhtml 被显示。当点击 Corfirm 按钮，显示状态页面 AddressStoredStatus.xhtml。当点击 Go Back 按钮，则第一个页面 AddressRegistration.xhtml 被显示。

程序清单 33-16 AddressRegistration.xhtml

```

1 <?xml version='1.0' encoding='UTF-8' ?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml"
5     xmlns:h="http://xmlns.jcp.org/jsf/html"
6     xmlns:f="http://xmlns.jcp.org/jsf/core">
7     <h:head>
8         <title>Student Registration Form</title>
9     </h:head>
10    <h:body>
11        <h:form>
12            <!-- Use h:graphicImage -->
13            <h3>Student Registration Form
14                <h:graphicImage name="usIcon.gif" library="image"/>
15            </h3>
16
17            Please register to your instructor's student address book.
18            <!-- Use h:panelGrid -->
19            <h:panelGrid columns="6">
20                <h:outputLabel value="Last Name" style="color:red"/>
21                <h:inputText id="lastNameInputText"
22                    value="#{addressRegistration.lastName}"/>
23                <h:outputLabel value="First Name" style="color:red"/>
24                <h:inputText id="firstNameInputText"
25                    value="#{addressRegistration.firstName}"/>
26                <h:outputLabel value="MI" />
27                <h:inputText id="miInputText" size="1"
28                    value="#{addressRegistration.mi}"/>
29            </h:panelGrid>
30
31            <h:panelGrid columns="4">
32                <h:outputLabel value="Telephone"/>
33                <h:inputText id="telephoneInputText"
34                    value="#{addressRegistration.telephone}"/>
35                <h:outputLabel value="Email"/>
36                <h:inputText id="emailInputText"
37                    value="#{addressRegistration.email}"/>
38            </h:panelGrid>
39
40            <h:panelGrid columns="4">
41                <h:outputLabel value="Street"/>
42                <h:inputText id="streetInputText"
43                    value="#{addressRegistration.street}"/>

```



```

44     </h:panelGrid>
45
46     <h:panelGrid columns="6">
47         <h:outputLabel value="City"/>
48         <h:inputText id="cityInputText"
49             value="#{addressRegistration.city}"/>
50         <h:outputLabel value="State"/>
51         <h:selectOneMenu id="stateSelectOneMenu"
52             value="#{addressRegistration.state}">
53             <f:selectItem itemLabel="Georgia-GA" itemValue="GA" />
54             <f:selectItem itemLabel="Oklahoma-OK" itemValue="OK" />
55             <f:selectItem itemLabel="Indiana-IN" itemValue="IN" />
56         </h:selectOneMenu>
57         <h:outputLabel value="Zip"/>
58         <h:inputText id="zipInputText"
59             value="#{addressRegistration.zip}"/>
60     </h:panelGrid>
61
62     <!-- Use command button -->
63     <h:commandButton value="Register"
64         action="#{addressRegistration.processSubmit()}" />
65     <br />
66     <h:outputText escape="false" style="color:red"
67         value="#{addressRegistration.requiredFields}" />
68 </h:form>
69 </h:body>
70 </html>

```

程序清单 33-17 ConfirmAddress.xhtml

```

1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5      xmlns:h="http://xmlns.jcp.org/jsf/html">
6      <h:head>
7          <title>Confirm Student Registration</title>
8      </h:head>
9      <h:body>
10         <h:form>
11             <h:outputText escape="false" style="color:red"
12                 value="#{registration1.input}" />
13             <h:panelGrid columns="2">
14                 <h:commandButton value="Confirm"
15                     action = "#{registration1.storeStudent()}" />
16                 <h:commandButton value="Go Back"
17                     action = "AddressRegistration" />
18             </h:panelGrid>
19         </h:form>
20     </h:body>
21 </html>

```

程序清单 33-18 AddressStoredStatus.xhtml

```

1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5      xmlns:h="http://xmlns.jcp.org/jsf/html">
6      <h:head>
7          <title>Address Stored?</title>
8      </h:head>
9      <h:body>
10         <h:form>
11             <h:outputText escape="false" style="color:green"

```

```
12         value="#{registration1.status}" />
13     </h:form>
14 </h:body>
15 </html>
```

程序清单 33-19 AddressRegistrationJSFBean.java

```
1  package jsf2demo;
2
3  import javax.inject.Named;
4  import javax.enterprise.context.SessionScoped;
5  import java.sql.*;
6  import java.io.Serializable;
7
8  @Named(value = "addressRegistration")
9  @SessionScoped
10 public class AddressRegistrationJSFBean implements Serializable {
11     private String lastName;
12     private String firstName;
13     private String mi;
14     private String telephone;
15     private String email;
16     private String street;
17     private String city;
18     private String state;
19     private String zip;
20     private String status = "Nothing stored";
21     // Use a prepared statement to store a student into the database
22     private PreparedStatement pstmt;
23
24     public AddressRegistrationJSFBean() {
25         initializeJdbc();
26     }
27
28     public String getLastName() {
29         return lastName;
30     }
31
32     public void setLastName(String lastName) {
33         this.lastName = lastName;
34     }
35
36     public String getFirstName() {
37         return firstName;
38     }
39
40     public void setFirstName(String firstName) {
41         this.firstName = firstName;
42     }
43
44     public String getMi() {
45         return mi;
46     }
47
48     public void setMi(String mi) {
49         this.mi = mi;
50     }
51
52     public String getTelephone() {
53         return telephone;
54     }
55
56     public void setTelephone(String telephone) {
57         this.telephone = telephone;
58     }
```

```
59
60 public String getEmail() {
61     return email;
62 }
63
64 public void setEmail(String email) {
65     this.email = email;
66 }
67
68 public String getStreet() {
69     return street;
70 }
71
72 public void setStreet(String street) {
73     this.street = street;
74 }
75
76 public String getCity() {
77     return city;
78 }
79
80 public void setCity(String city) {
81     this.city = city;
82 }
83
84 public String getState() {
85     return state;
86 }
87
88 public void setState(String state) {
89     this.state = state;
90 }
91
92 public String getZip() {
93     return zip;
94 }
95
96 public void setZip(String zip) {
97     this.zip = zip;
98 }
99
100 private boolean isRequiredFieldsFilled() {
101     return !(lastName == null || firstName == null
102         || lastName.trim().length() == 0
103         || firstName.trim().length() == 0);
104 }
105
106 public String processSubmit() {
107     if (isRequiredFieldsFilled())
108         return "ConfirmAddress";
109     else
110         return "";
111 }
112
113 public String getRequiredFields() {
114     if (isRequiredFieldsFilled())
115         return "";
116     else
117         return "Last Name and First Name are required";
118 }
119
120 public String getInput() {
121     return "<p style='color:red'>You entered <br />"
122         + "Last Name: " + lastName + "<br />"
123         + "First Name: " + firstName + "<br />"
```

```

124         + "MI: " + mi + "<br />"
125         + "Telephone: " + telephone + "<br />"
126         + "Email: " + email + "<br />"
127         + "Street: " + street + "<br />"
128         + "City: " + city + "<br />"
129         + "Street: " + street + "<br />"
130         + "City: " + city + "<br />"
131         + "State: " + state + "<br />"
132         + "Zip: " + zip + "</p>";
133     }
134
135     /** Initialize database connection */
136     private void initializeJdbc() {
137         try {
138             // Explicitly load a MySQL driver
139             Class.forName("com.mysql.jdbc.Driver");
140             System.out.println("Driver loaded");
141
142             // Establish a connection
143             Connection conn = DriverManager.getConnection(
144                 "jdbc:mysql://localhost/javabook", "scott", "tiger");
145
146             // Create a Statement
147             pstmt = conn.prepareStatement("insert into Address (lastName,"
148                 + " firstName, mi, telephone, email, street, city, "
149                 + "state, zip) values (?, ?, ?, ?, ?, ?, ?, ?, ?)");
150         }
151         catch (Exception ex) {
152             System.out.println(ex);
153         }
154     }
155
156     /** Store an address to the database */
157     public String storeStudent() {
158         try {
159             pstmt.setString(1, lastName);
160             pstmt.setString(2, firstName);
161             pstmt.setString(3, mi);
162             pstmt.setString(4, telephone);
163             pstmt.setString(5, email);
164             pstmt.setString(6, street);
165             pstmt.setString(7, city);
166             pstmt.setString(8, state);
167             pstmt.setString(9, zip);
168             pstmt.executeUpdate();
169             status = firstName + " " + lastName
170                 + " is now registered in the database.";
171         }
172         catch (Exception ex) {
173             status = ex.getMessage();
174         }
175
176         return "AddressStoredStatus";
177     }
178
179     public String getStatus() {
180         return status;
181     }
182 }

```

一个会话范围内的受管 bean 必须实现 `java.io.Serializable` 接口。因此, `AddressRegistration` 类定义为 `java.io.Serializable` 的子类型。

`AddressRegistration` 这个 JSF 页面中 Register 按钮的触发动作是 `processSubmit()`

(AddressRegistration.xhtml 中第 64 行)。该方法检查姓和名是否不为空 (AddressRegistrationJSFBean.java 中第 106 ~ 111 行)。如果不为空, 则返回字符串 "ConfirmAddress", 这将导致 ConfirmAddress 的 JSF 页面被显示。

ConfirmAddress 这个 JSF 页面显示用户输入的数据 (ConfirmAddress.xhtml 第 12 行)。getInput() 方法 (AddressRegistrationJSFBean.java 中第 120 ~ 133 行) 得到输入。

ConfirmAddress 这个 JSF 页面中 Confirm 按钮的动作是 storeStudent() (ConfirmAddress.xhtml 中第 15 行)。该方法将地址存储到数据库中 (AddressRegistrationJSFBean.java 中第 157 ~ 177 行), 并返回字符串 "AddressStoredStatus", 这将导致显示 AddressStoredStatus 页面。状态信息显示在该页面中 (AddressStoredStatus.xhtml 中第 12 行)。

ConfirmAddress 页面中 Go Back 按钮的动作是 "AddressRegistration" (ConfirmAddress.xhtml 中第 17 行)。这将导致显示 AddressRegistration 页面, 让用户可以重新输入。

受管 bean 的范围是会话 (AddressRegistrationJSFBean.java 中第 9 行), 因此多个页面可以共享同一个 bean。

注意, 该程序显式装载数据库驱动 (AddressRegistrationJSFBean.java 中第 139 行)。有时候, 诸如 NetBeans 这样的 IDE 无法找到一个合适的驱动。显式装载一个驱动可以避免这个问题。

关键技术语

application scope (应用范围)

JavaBean (JavaBean 组件)

request scope (请求范围)

scope (范围)

session scope (会话范围)

view scope (视图范围)

本章小结

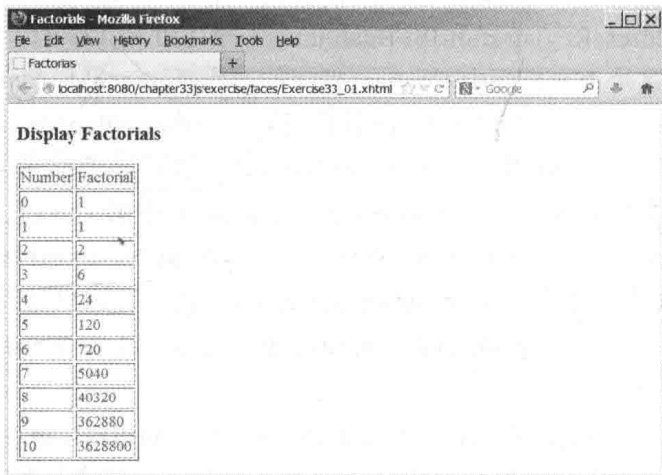
1. JSF 使得 Java 代码和 HTML 完全分离。
2. facelet 是混合使用了 JSF 标签和 XHTML 标签的 XHTML 页面。
3. JSF 应用使用模型 - 视图 - 控制器 (MVC) 架构实现, 这样将应用程序数据 (包含在模型中) 和图形表示 (视图) 进行了分离。
4. 控制器是负责协调视图和模型之间交互的 JSF 框架。
5. JSF 中, facelet 是表现数据的视图。数据从 Java 对象处得到。使用 Java 类定义对象。
6. JSF 中, 从 facelet 访问的对象是 JavaBean 对象。
7. JSF 表达式可以通过属性名字, 或者调用方法来获得当前时间。
8. JSF 提供许多元素用来显示 GUI 组件。以 h 为前缀的标签在 JSF HTML 标签库中。以 f 为前缀的标签在 JSF 核心标签库中。
9. 可以指定 JavaBean 对象的范围在应用范围、会话范围、视图范围或者请求范围内。
10. 只要停留在一个视图上, 视图范围将保持 bean 有效。视图范围介于会话范围和请求范围之间。
11. JSF 提供了一些方便且强大的方式来进行输入验证。可以使用 JSF 核心标签库中的标准验证器, 也可以创建自定义验证器。

测试题

回答位于网址 www.cs.armstrong.edu/liang/intro10e/quiz.html 的本章测试题。

编程练习题

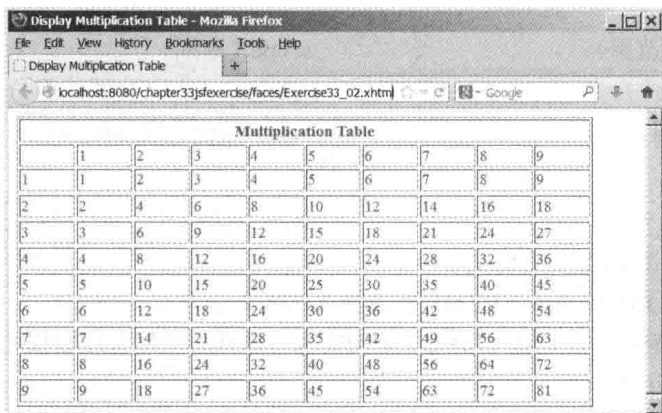
- *33.1 (JSF 中的阶乘表) 编写一个 JSF 页面, 如图 33-25 显示一个阶乘页面。在一个 `h:outputText` 组件中显示表格。将其 `escape` 属性设置为 `false`, 从而将其显示为 HTML 内容。



Number	Factorial
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800

图 33-25 JSF 页面为 0 到 10 之间的数字在表格中显示阶乘

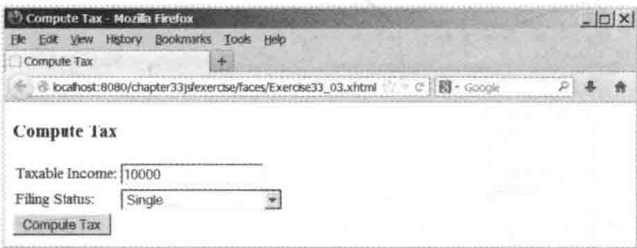
- *33.2 (乘法表) 编写一个 JSF 页面, 如图 33-26 显示一个乘法表。



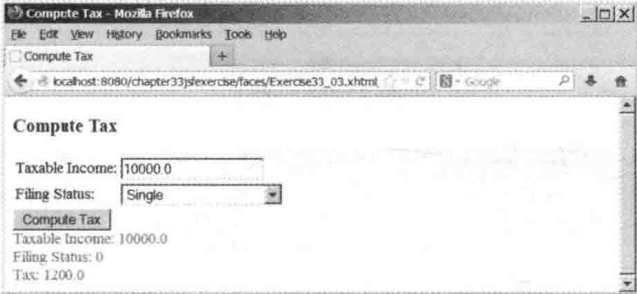
	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

图 33-26 JSF 页面显示乘法表

- *33.3 (计算税) 编写一个 JSF 页面, 让用户输入需要缴税的收入以及婚姻状况, 如图 33-27a 所示。点击 `Compute Tax` 按钮计算并显示缴税, 如图 33-27b 所示。使用程序清单 3-5 中引入的 `computeTax` 方法来计算税。
- *33.4 (计算贷款) 编写一个 JSF 页面, 让用户输入贷款额度、利率以及年数, 如图 33-28a 所示。点击 `Compute Loan Payment` 按钮计算并显示每个月以及整个的贷款支付, 如图 33-28b 所示。使用程序清单 10-2 中给出的 `Loan` 类来计算每个月以及整个的支付。
- *33.5 (加法测试) 编写一个 JSF 页面, 可以随机生成加法测试题, 如图 33-29a 所示。当用户回答完所有的题目, 如图 33-29b 显示结果。
- *33.6 (大的阶乘) 重写编程练习题 33.1, 使之可以处理大的阶乘。使用 10.9 节中介绍的 `BigInteger` 类。
- *33.7 (猜测生日) 程序清单 4-3 给出了一个猜测生日的程序。编写一个 JSF 程序, 显示 5 个数据集, 如图 33-30a 所示。用户勾选合适选项并且点击 `Guess Birthday` 按钮后, 程序如图 33-30b 显示生日。

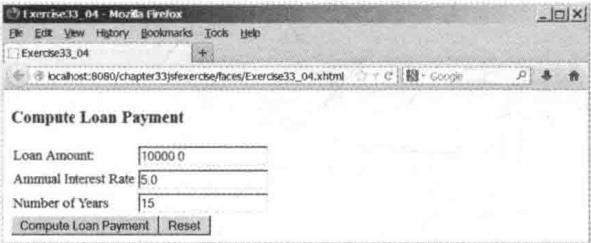


a)



b)

图 33-27 JSF 页面计算税

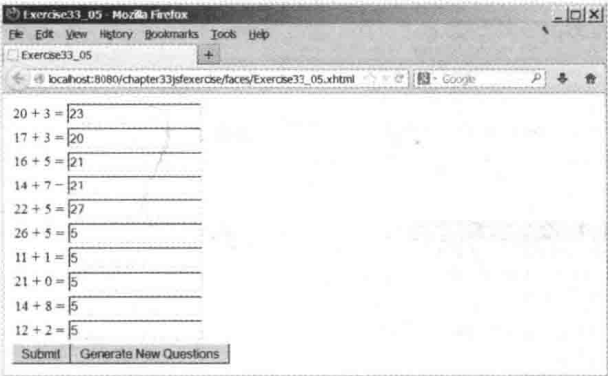


a)



b)

图 33-28 JSF 页面计算借贷支付



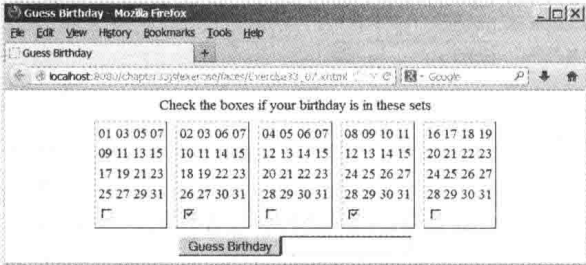
a)

图 33-29 程序在 a 中显示加法问题，在 b 中显示答案

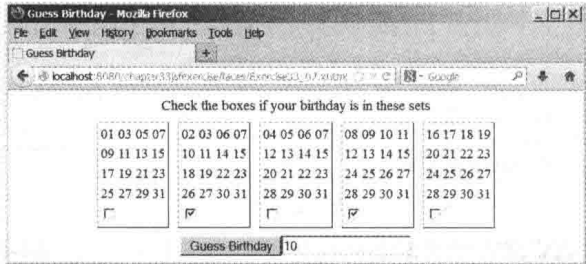


b)

图 33-29 (续)



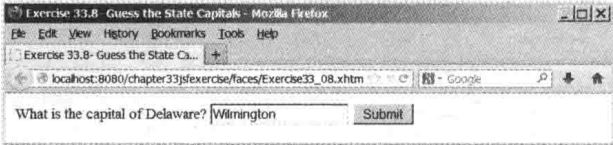
a)



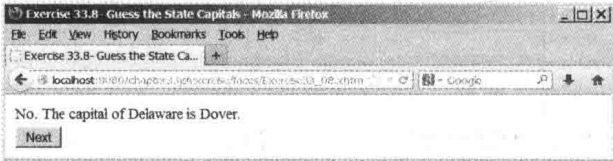
b)

图 33-30 a) 程序显示 5 组数字，让用户勾选；b) 程序显示日期

*33.8 (猜测首府) 编写一个 JSF 页面，提示用户输入一个州的首府，如图 33-31a 所示。得到用户输入后，程序报告答案是否正确，如图 33-31b 所示。可以点击 Next 按钮显示另外一个问题。可以使用二维数组来存储州和首府，如编程练习题 8.37 所提示。从数组创建一个线性表，并引用 shuffle 方法来重新对线性表排序从而问题将以随机顺序显示。



a)



b)

图 33-31 a) 程序显示一个问题；b) 程序显示问题的答案

*33.9（访问和更新 staff 表）编写一个 JSF 程序，可以观看、插入以及更新保存在数据库中的员工信息，如图 33-32 所示。View 按钮显示一个指定 ID 的记录。Staff 表如下创建：

```
create table Staff (  
    id char(9) not null,  
    lastName varchar(15),  
    firstName varchar(15),  
    mi char(1),  
    address varchar(20),  
    city varchar(20),  
    state char(2),  
    telephone char(10),  
    email varchar(40),  
    primary key (id)  
);
```

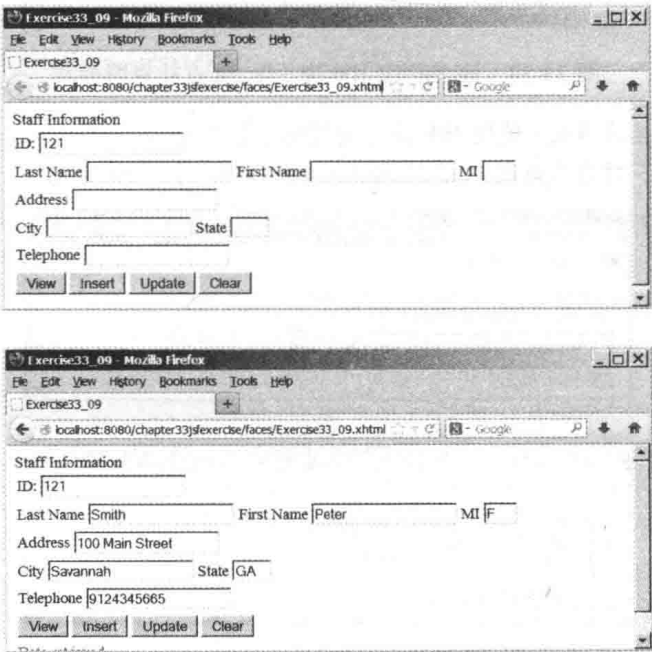


图 33-32 Web 页面让你观看、插入以及更新员工信息

*33.10（随机扑克牌）编写一个 JSF 程序，显示一叠 52 张扑克牌中的 4 张随机扑克牌如图 33-33 所示。当用户点击 Refresh 按钮时候，4 张新的随机扑克牌被显示。

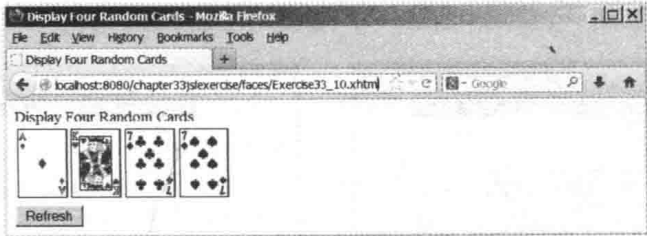


图 33-33 该 JSF 应用显示 4 张随机扑克牌

***33.11（游戏：24 点扑克牌游戏）使用 JSF 重写编程练习题 20.13，如图 33-34 所示。当点击 Refresh 按钮时，程序显示 4 张随机扑克牌，并且显示一个 24 点的解答，如果该解答存在。否则，显示 No solution。

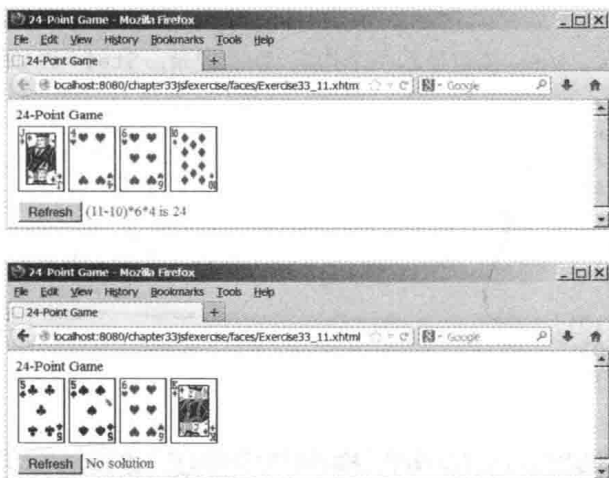


图 33-34 该 JSF 应用解决一个 24 点扑克牌游戏

***33.12 (游戏: 24 点扑克牌游戏) 使用 JSF 重写编程练习题 20.17, 如图 33-35 所示。程序让用户输入 4 张扑克牌的值, 并且当点击 Find a Solution 按钮时找到一个解决方案。

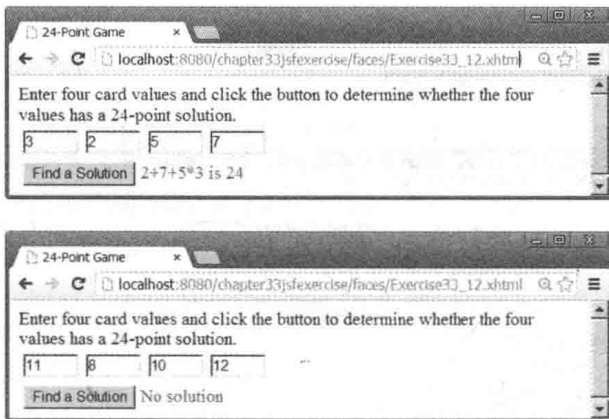


图 33-35 用户输入 4 个数字, 程序找到一个解答

*33.13 (一个星期中的周几) 编写一个程序, 对于一个给定的年、月、日, 显示是一个星期中的周几, 如图 33-36 所示。程序让用户选择一个日期、月份以及年份, 然后点击 Get Day of Week 按钮来显示一个星期中的周几。如果这是将来的某一天则 Time 字段显示 Future, 否则显示 Past。使用蔡勒公式来找到一个星期中的某一天 (参见编程练习题 3.21)。

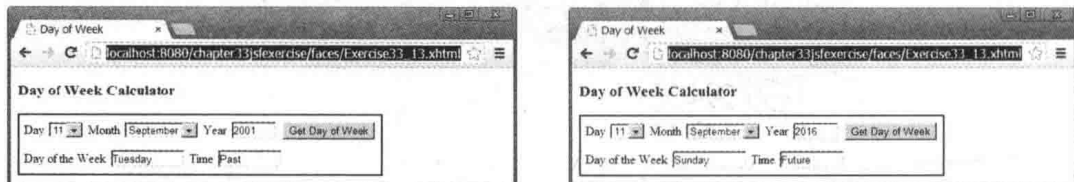


图 33-36 用户输入一个年、月、日, 程序找到属于一个星期中的周几

Java 关键字

下面是 Java 语言保留使用的 50 个关键字：

abstract	double	int	super
assert	else	interface	switch
boolean	enum	long	synchronized
break	extends	native	this
byte	final	new	throw
case	finally	package	throws
catch	float	private	transient
char	for	protected	try
class	goto	public	void
const	if	return	volatile
continue	implements	short	while
default	import	static	
do	instanceof	strictfp [⊖]	

关键字 `goto` 和 `const` 是 C++ 保留的关键字，目前并没有在 Java 中使用到。如果它们出现在 Java 程序中，Java 编译器能够识别它们，并产生错误信息。

字面常量 `true`、`false` 和 `null` 如同字面值 `100` 一样，不是关键字。但是它们也不能用作标识符，就像 `100` 不能用作标识符一样。

在代码清单中，我们对 `true`、`false` 和 `null` 使用了关键字的颜色，以和 Java IDE 中它们的颜色保持一致。

⊖ `strictfp` 关键字是用于修饰方法或者类的，使其使用严格的浮点计算。浮点计算可以使用以下两种模式：严格的和非严格的。严格模式可以保证计算结果在所有的虚拟机实现中都是一样的。非严格模式允许计算的中间结果以一种扩展的格式存储，该格式不同于标准的 IEEE 浮点数格式。扩展格式是依赖于机器的，可以使代码执行更快。然而，当在不同的虚拟机上使用非严格模式执行代码时，可能不会总能精确地得到同样结果。默认情况下，非严格模式被用于浮点数的计算。若在方法和类中使用严格模式，需要在方法或者类的声明前面增加 `strictfp` 关键字。严格的浮点数可能会比非严格浮点数具有略好的精确度，但这种区别仅影响部分应用。严格模式不会被继承，即，在类或者接口的声明中使用 `strictfp` 不会使得继承的子类或接口也是严格模式。

ASCII 字符集

表 B-1 和表 B-2 分别列出了 ASCII 字符与它们相应的十进制和十六进制编码。字符的十进制或十六进制编码是字符行下标和列下标的组合。例如，在表 B-1 中，字母 A 在第 6 行第 5 列，所以它的十进制代码为 65；在表 B-2 中，字母 A 在第 4 行第 1 列，所以它的十六进制代码为 41。

表 B-1 十进制编码的 ASCII 字符集

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dcl	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	'
4	()	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[\]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

表 B-2 十六进制编码的 ASCII 字符集

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht	nl	vt	ff	cr	so	si
1	dle	dcl	dc2	dc3	dc4	nak	syn	etb	can	em	sub	esc	fs	gs	rs	us
2	sp	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	del

操作符优先级表

操作符按照优先级递减的顺序从上到下列出。同一栏中的操作符优先级相同，它们的结合方向如表中所示。

操作符	名称	结合方向	操作符	名称	结合方向
()	圆括号	从左向右	>>>	用零扩展的右移	从左向右
()	函数调用	从左向右	<	小于	从左向右
[]	数组下标	从左向右	<=	小于等于	从左向右
.	对象成员访问	从左向右	>	大于	从左向右
++	后置增量	从右向左	>=	大于等于	从左向右
--	后置减量	从右向左	instanceof	检测对象类型	从左向右
++	前置增量	从右向左	==	相等	从左向右
--	前置减量	从右向左	!=	不等	从左向右
+	一元加	从右向左	&	(无条件与)	从左向右
-	一元减	从右向左	^	(异或)	从左向右
!	一元逻辑非	从右向左		(无条件或)	从左向右
(type)	一元类型转换	从右向左	&&	条件与	从左向右
new	创建对象	从右向左		条件或	从左向右
*	乘法	从左向右	?:	三元条件	从右向左
/	除法	从左向右	=	赋值	从右向左
%	求余	从左向右	+=	加法赋值	从右向左
+	加法	从左向右	-=	减法赋值	从右向左
-	减法	从左向右	*=	乘法赋值	从右向左
<<	左移	从左向右	/=	除法赋值	从右向左
>>	用符号位扩展的右移	从左向右	%=	求余赋值	从右向左

Java 修饰符

修饰符用于类和类的成员（构造方法、方法、数据和类一级的块），但 `final` 修饰符也可以用在方法中的局部变量上。可以用在类上的修饰符称为类修饰符（`class modifier`）。可以用在方法上的修饰符称为方法修饰符（`method modifier`）。可以用在数据域上的修饰符称为数据修饰符（`data modifier`）。可以用在类一级块上的修饰符称为块修饰符（`block modifier`）。下表给出 Java 修饰符的一个总结。

修饰符	类	构造方法	方法	数据	块	解释
(default) [⊖]	✓	✓	✓	✓	✓	类、构造方法、方法或数据域在所在的包中可见
public	✓	✓	✓	✓		类、构造方法、方法或数据域在任何包任何程序中都可见
private		✓	✓	✓		构造方法、方法或数据域只在所在的类中可见
protected		✓	✓	✓		构造方法、方法或数据域在所属包中可见，或者在任何包中该类的子类中可见
static			✓	✓	✓	定义类方法、类数据域或静态初始化模块
final	✓		✓	✓		终极类不能扩展。终极方法不能在子类中修改。终极数据域是常量
abstract	✓		✓			抽象类必须被扩展。抽象方法必须在具体的子类中实现
native			✓			用 <code>native</code> 修饰的方法表明它是用 Java 以外的语言实现的
synchronized			✓		✓	同一时间只有一个线程可以执行这个方法
strictfp	✓		✓			使用精确浮点数计算模式，保证在所有的 Java 虚拟机中计算结果都相同
transient				✓		标记实例数据域，使其不进行序列化

默认（没有修饰符）、`public`、`private` 以及 `protected` 等修饰符称为可见或者可访问性修饰符，因为它们给定了类，以及类的成员是如何被访问的。

`public`、`private`、`protected`、`static`、`final` 以及 `abstract` 也可以用于内部类。

⊖ 默认访问没有任何修饰符与之关联。例如：`class Test{}`

特殊浮点值

整数除以零是非法的，会抛出异常 `ArithmeticException`，但是浮点值除以零不会引起异常。在浮点运算中，如果运算结果对 `double` 型或 `float` 型来说数值太大，则向上溢出为无穷大；如果运算结果对 `double` 型或 `float` 型来说数值太小，则向下溢出为零。Java 用特殊的浮点值 `POSITIVE_INFINITY`、`NEGATIVE_INFINITY` 和 `NaN` (`Not a Number`，非数值) 来表示这些结果。这些值被定义为 `Float` 类和 `Double` 类中的特殊常量。

如果正浮点数除以零，结果为 `POSITIVE_INFINITY`。如果负浮点数除以零，结果为 `NEGATIVE_INFINITY`。如果浮点数零除以零，结果为 `NaN`，表示这个结果在数学上是无定义的。这三个值的字符串表示为 `Infinity`、`-Infinity` 和 `NaN`。例如，

```
System.out.print(1.0 / 0); // Print Infinity
System.out.print(-1.0 / 0); // Print -Infinity
System.out.print(0.0 / 0); // Print NaN
```

这些特殊值也可以在运算中用作操作数。例如，一个数除以 `POSITIVE_INFINITY` 得到零。表 E-1 总结了运算符 `/`、`*`、`%`、`+` 和 `-` 的各种组合。

表 E-1 特殊的浮点值

x	y	x/y	x*y	x%y	x+y	x-y
Finite	± 0.0	± infinity	± 0.0	NaN	Finite	Finite
Finite	± infinity	± 0.0	± 0.0	x	± infinity	infinity
± 0.0	± 0.0	NaN	± 0.0	NaN	± 0.0	± 0.0
± infinity	Finite	± infinity	± 0.0	NaN	± infinity	± infinity
± infinity	± infinity	NaN	± 0.0	NaN	± infinity	infinity
± 0.0	± infinity	± 0.0	NaN	± 0.0	± infinity	± 0.0
NaN	Any	NaN	NaN	NaN	NaN	NaN
Any	NaN	NaN	NaN	NaN	NaN	NaN

注意：如果一个操作数是 `NaN`，则结果一定是 `NaN`。

数 系

F.1 引言

因为计算机本身只能存储和处理 0 和 1，所以其内部使用的是二进制数。二进制数系只有两个数：0 和 1。在计算机中，数字或字符是以由 0 和 1 组成的序列来存储的。每个 0 或 1 都称为一个比特（二进制数字）。

我们在日常生活中使用十进制数。当我们在程序中编写一个数字，如 20，它被假定为一个十进制数。在计算机内部，通常会用软件将十进制数转换成二进制数，反之亦然。

我们使用十进制数编写程序。然而，如果要与操作系统打交道，需要使用二进制数以达到“机器级”。二进制数冗长烦琐，所以经常使用十六进制数简化二进制数，每个十六进制数可以确切表示四个二进制数。十六进制数系有十六个数：0 ~ 9、A ~ F，其中字母 A、B、C、D、E 和 F 对应十进制数 10、11、12、13、14 和 15。

十进制数系中的数是 0、1、2、3、4、5、6、7、8 和 9。一个十进制数是用一个或多个这些数所构成的一个序列来表示的。这个序列中每个数所表示的值和它的位置有关，序列中数的位置决定了 10 的幂次。例如，十进制数 7423 中的数 7、4、2 和 3 分别表示 7000、400、20 和 3，如下所示：

$$\boxed{7} \boxed{4} \boxed{2} \boxed{3} = 7 \times 10^3 + 4 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

$$10^3 10^2 10^1 10^0 = 7000 + 400 + 20 + 3 = 7423$$

十进制数系有十个数，它们的位置值都是 10 的整数次幂。10 是十进制数系的基数。类似地，由于二进制数系有两个数，所以它的基数为 2；而十六进制数系有 16 个数，所以它的基数为 16。

如果 1101 是一个二进制数，那么数 1、1、0 和 1 分别表示：

$$\boxed{1} \boxed{1} \boxed{0} \boxed{1} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$2^3 2^2 2^1 2^0 = 8 + 4 + 0 + 1 = 13$$

如果 7423 是一个十六进制数，那么数字 7、4、2 和 3 分别表示：

$$\boxed{7} \boxed{4} \boxed{2} \boxed{3} = 7 \times 16^3 + 4 \times 16^2 + 2 \times 16^1 + 3 \times 16^0$$

$$16^3 16^2 16^1 16^0 = 28672 + 1024 + 32 + 3 = 29731$$

F.2 二进制数与十进制数之间的转换

给定二进制数 $b_n b_{n-1} b_{n-2} \cdots b_2 b_1 b_0$ ，等价的十进制数为

$$b_n \times 2^n + b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

下面是二进制数转换为十进制数的例子：

二进制	转换公式	十进制
10	$1 \times 2^1 + 0 \times 2^0$	2
1000	$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$	8
10101011	$1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$	171

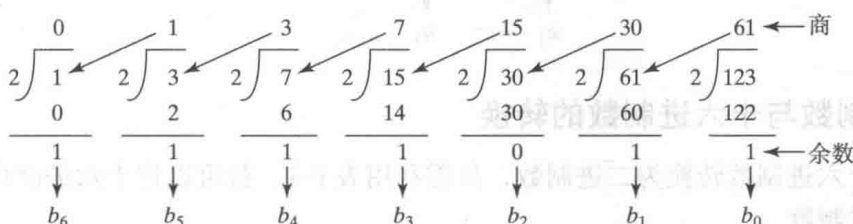
把一个十进制数 d 转换为二进制数，就是求满足

$$d = b_n \times 2^n + b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

的位 $b_n, b_{n-1}, b_{n-2}, \cdots, b_2, b_1$ 和 b_0 。

用 2 不断地除 d ，直到商为 0 为止，余数即为所求的位 $b_0, b_1, \cdots, b_{n-2}, b_{n-1}, b_n$ 。

例如，十进制数 123 用二进制数 1111011 表示，所做的转换如下：



提示：Windows 操作系统所带的计算器是进行数制转换的一个有效工具，如图 F-1 所示。

要运行它，从 Start 按钮搜索 Calculator 并运行 Calculator，然后在 View 菜单下面选择 Scientific。



图 F-1 使用 Windows 的计算器进行数制转换

F.3 十六进制数与十进制数的转换

给定十六进制数 $h_n h_{n-1} h_{n-2} \cdots h_2 h_1 h_0$ ，其等价的十进制数为

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \cdots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

下面是十六进制数转换为十进制数的例子：

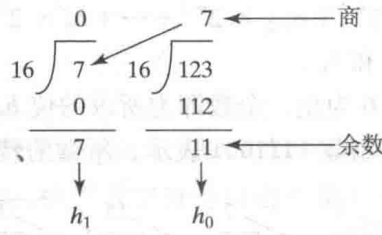
十六进制	转换公式	十进制
7F	$7 \times 16^1 + 15 \times 16^0$	127
FFFF	$15 \times 16^3 + 15 \times 16^2 + 15 \times 16^1 + 15 \times 16^0$	65535
431	$4 \times 16^2 + 3 \times 16^1 + 1 \times 16^0$	1073

将一个十进制数 d 转换为十六进制数，就是求满足

$$d = h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \cdots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

的位 $h_n, h_{n-1}, h_{n-2}, \cdots, h_2, h_1$ 和 h_0 。用 16 不断地除 d ，直到商为 0 为止。余数即为所求的位 $h_0, h_1, \cdots, h_{n-2}, h_{n-1}, h_n$ 。

例如，十进制数 123 用十六进制表示为 7B，所做的转换如下：



F.4 二进制数与十六进制数的转换

将一个十六进制数转换为二进制数，只需利用表 F-1，就可以把十六进制数的每一位转换为四位二进制数。

例如，十六进制数 7B 转换为二进制是 1111011，其中 7 的二进制表示为 111，B 的二进制表示为 1011。

要将一个二进制数转换为十六进制数，从右向左将每四位二进制数转换为一位十六进制数。

例如，二进制数 1110001101 的十六进制表示是 38D，因为 1101 是 D，1000 是 8，11 是 3，如下所示：

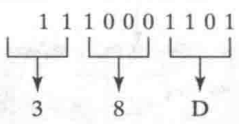


表 F-1 十六进制数转换为二进制数

十六进制	二进制	十进制	十六进制	二进制	十进制
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	A	1010	10
3	0011	3	B	1011	11
4	0100	4	C	1100	12
5	0101	5	D	1101	13
6	0110	6	E	1110	14
7	0111	7	F	1111	15

注意：八进制数也很有用。八进制数系有 0 到 7 共八个数。十进制数 8 在八进制数系中的作用就和十进制数系中的 10 一样。

这里有一些好的在线资源，用于练习数值转换：

- http://forums.cisco.com/CertCom/game/binary_game_page.htm
- <http://people.sinclair.edu/nickreeder/Flash/binDec.htm>
- <http://people.sinclair.edu/nickreeder/Flash/binHex.htm>

复习题

F.1 将下列十进制数转换为十六进制数和二进制数。

100; 4340; 2000

F.2 将下列二进制数转换为十六进制数和十进制数。

1000011001; 100000000; 100111

F.3 将下列十六进制数转换为二进制数和十进制数。

FEFA9; 93; 2000

位 操 作

用机器语言编写程序，经常要直接处理二进制数值，并在位级别上执行操作。Java 提供了位操作符和移位操作符，如表 G-1 所示。

表 G-1

操作符	名称	示例（例中使用字节）	描述
&	位与	10101110 & 10010010 得到 10000010	两个相应位上的比特如果都为 1，则执行与操作会得到 1
	位或	10101110 10010010 得到 10111110	两个相应位上的比特如果其中有一个为 1，则执行或操作会得到 1
^	位与或	10101110 ^ 10010010 得到 00111100	两个相应位上的比特如果相异，则执行与或操作会得到 1
~	求反	~10101110 得到 01010001	操作符将每个比特从 0 到 1 或者从 1 到 0 进行转换
<<	左移位	10101110 << 2 得到 10111000	操作符将其左边的操作数按照第二个操作数指定的位移数进行左移位，右边空出来的补 0
>>	带符号位右移位	10101110 >> 2 得到 11101011 00101110 >> 2 得到 00001011	操作符将其第一个操作数按照第二个操作数指定的位移数进行右移位，最高位补上符号位
>>>	无符号位右移位	10101110 >>> 2 得到 00101011 00101110 >>> 2 得到 00001011	操作符将其第一个操作数按照第二个操作数指定的位移数进行右移位，左边空出来的补 0

位操作符仅适用于整数类型（byte、short、int 和 long）。位操作涉及的字符将转换为整数。所有的位操作符可以构成位赋值操作符，例如 =，|=，<<=，>>=，以及 >>>=。

正则表达式

经常会需要编写代码来验证用户输入，比如验证输入是否是一个数字，是否是一个全部小写的字符串，或者社会安全号。如何编写这种类型的代码呢？一个简单而有效的做法是使用正则表达式来完成这个任务。

正则表达式 (regular expression, 简称为 regex) 是一个字符串，用来描述匹配一个字符串集合的模式。对于字符串处理来说，正则表达式是一个强大的工具。可以使用正则表达式来匹配、替换和分割字符串。

H.1 匹配字符串

让我们从 String 类中的 matches 方法开始。乍一看，matches 方法很类似 equals 方法。例如，以下两个语句结果都为 true。

```
"Java".matches("Java");  
"Java".equals("Java");
```

然而，matches 方法更强大。它不仅可以匹配固定字符串，还可以匹配一个模式的字符串集。例如，以下语句结果都为 true。

```
"Java is fun".matches("Java.*")  
"Java is cool".matches("Java.*")  
"Java is powerful".matches("Java.*")
```

前面语句中的 "Java.*" 是一个正则表达式。它描述了一个字符串模式，以 Java 开始，后面跟 0 个或者多个字符串。这里，子字符串 .* 匹配任何 0 个或者多个字符。

H.2 正则表达式语法

正则表达式由面值字符和特殊符号组成。表 H-1 列出了正则表达式常用的语法。

{ } 注意：反斜杠是一个特殊的字符，在字符串中开始转义序列。因此 Java 中需要使用 \\d 来表示 \d。

{ } 注意：回顾下，空白字符是 ' '、'\t'、'\n'、'\r'，或者 '\f'。因此，\s 和 [\t\n\r\f] 等同，\S 和 [^\t\n\r\f] 等同。

表 H-1 常用的正则表达式

正则表达式	匹配	示例
x	指定字符 x	Java 匹配 Java
.	任意单个字符	Java 匹配 J..a
(ab cd)	ab 或者 cd	ten 匹配 t(en im)
[abc]	a、b 或者 c	Java 匹配 Ja[uvw]a
[^abc]	除开 a、b 或者 c 外的任意字符	Java 匹配 Ja[^ars]a
[a-z]	a 到 z	Java 匹配 [A-M]av[a-d]
[^a-z]	除开 a 到 z 的任意字符	Java 匹配 Jav[^b-d]

(续)

正则表达式	匹配	示例
[a-e[m-p]]	a 到 e 或 m 到 p	Java 匹配 [A-G[I-M]]av[a-d]
[a-e&&[c-p]]	a 到 e 与 c 到 p 的交集	Java 匹配 [A-P&&[I-M]]av[a-d]
\d	个位数, 等同于 [0-9]	Java2 匹配 "Java[\\d]"
\D	一位非数字	\$Java 匹配 "[\\D][\\D]ava"
\w	单词字符	Java1 匹配 "[\\w]ava[\\w]"
\W	非单词字符	\$Java 匹配 "[\\W][\\w]ava"
\s	空白字符	"Java 2" 匹配 "Java\\s2"
\S	非空白字符	Java 匹配 "[\\S]ava"
p*	模式 p 的 0 或者多次出现	aaaabb 匹配 "a*bb" ababab 匹配 "(ab)*"
p+	模式 p 的 1 或者多次出现	a 匹配 "a+b*" able 匹配 "(ab)+.*"
p?	模式 p 的 0 或者 1 次出现	Java 匹配 "J?Java" Java 匹配 "J?ava"
p{n}	模式 p 的正好 n 次出现	Java 匹配 "Ja{1}.*" Java 不匹配 ".{2}"
p{n,}	模式 p 的至少 n 次出现	aaaa 匹配 "a{1,}" a 不匹配 "a{2,}"
p{n,m}	模式 p 出现次数位于 n 和 m 间 (不包含)	aaaa 匹配 "a{1,9}" abb 不匹配 "a{2,9}bb"

注意：单词字符是任何的字母，数字或者下划线字符。因此 \w 等同于 [a-zA-Z][0-9_] 或者简化为 [a-Za-z0-9_]。 \W 等同于 [^a-Za-z0-9_]。

注意：表 H-1 中最后六个实体 *、+、?、{n}、{n,}，以及 {n, m} 称为量词符，用于确定量词符前面的模式会重复多少次。例如，A* 匹配 0 或者多个 A，A+ 匹配 1 或者多个 A，A? 匹配 0 或者 1 个 A。A{3} 精确匹配 AAA，A{3,} 匹配至少 3 个 A，A{3,6} 匹配 3 到 6 之间个 A。* 等同于 {0,}，+ 等同于 {1,}，? 等同于 {0,1}。

警告：不要在重复量词符中使用空白。例如，A{3,6} 不能写成逗号后面有一个空白符的 A{3, 6}。

注意：可以使用括号来将模式进行分组。例如，(ab){3} 匹配 ababab，但是 ab{3} 匹配 abbb。

让我们用一些示例来演示如何构建正则表达式。

1. 示例 1

社会安全号的模式是 xxx-xx-xxx，其中 x 是一位数字。社会安全号的正则表达式可以描述为

[\\d]{3}-[\\d]{2}-[\\d]{4}

例如

"111-22-3333".匹配 ("[\\d]{3}-[\\d]{2}-[\\d]{4}") 返回 true.
"11-22-3333".匹配 ("[\\d]{3}-[\\d]{2}-[\\d]{4}") 返回 false.

2. 示例 2

偶数以数字 0、2、4、6 或者 8 结尾。偶数的模式可以描述为

```
[\\d]*[02468]
```

例如,

```
"123".matches("[\\d]*[02468]") 返回 false.  
"122".matches("[\\d]*[02468]") 返回 true.
```

3. 示例 3

电话号码的模式是 (xxx)xxx-xxxx, 这里 x 是一位数字, 并且第一位数字不能为 0。电话号码的正则表达式可以描述为

```
\\([1-9][\\d]{2}\\) [\\d]{3}-[\\d]{4}
```

注意: 括号 (和) 在正则表达式中是特殊字符, 用于对模式分组。为了在正则表达式中表示字面值 (或者), 必须使用 \\(和 \\)。

例如

```
"(912) 921-2728".matches("\\([1-9][\\d]{2}\\) [\\d]{3}-[\\d]{4}")  
返回 true.  
"921-2728".matches("\\([1-9][\\d]{2}\\) [\\d]{3}-[\\d]{4}")  
返回 false.
```

4. 示例 4

假定姓由最多 25 个字母组成, 并且第一个字母为大写形式。则姓的模式可以描述为

```
[A-Z][a-zA-Z]{1,24}
```

注意: 不能任意放空白符到正则表达式中。如 [A-Z][a-Za-z]{1,24} 将报错。例如:

```
"Smith".matches("[A-Z][a-zA-Z]{1,24}") 返回 true.  
"Jones123".matches("[A-Z][a-zA-Z]{1,24}") 返回 false.
```

5. 示例 5

Java 标识符在第 2.4 节中定义

- 标识符必须以字母、下划线 (_), 或者美元符号 (\$) 开始。不能以数字开头。
- 标识符是一个由字母、数字、下划线 (_) 和美元符号组成的字符序列。

标识符的模式可以描述为

```
[a-zA-Z_$][\\w$]*
```

6. 示例 6

什么字符串匹配正则表达式 "Welcome to (Java|HTML)" ? 答案是 Welcome to Java 或者 Welcome to HTML。

7. 示例 7

什么字符串匹配正则表达式 "A.*" ? 答案是任何以字母 A 开头的字符串。

H.3 替换和分割字符串

如果字符串匹配正则表达式, String 类的 matches 方法返回 true。String 类也包含 replaceAll、replaceFirst 和 split 方法, 用于替换和分割字符串, 如图 H-1 所示。

replaceAll 方法替换所有匹配的子字符串, replaceFirst 方法替换第一个匹配的子字符串。例如, 下面代码

```
System.out.println("Java Java Java".replaceAll("\\w\\w", "wi"));
```

java.lang.String	
<code>+matches(regex: String): boolean</code> <code>+replaceAll(regex: String, replacement: String): String</code>	如果字符串匹配模式，则返回 true 将匹配的子字符串替换为 replacement 变量中的字符串，并返回新的字符串
<code>+replaceFirst(regex: String, replacement: String): String</code>	将匹配的个子字符串替换为 replacement 变量中的字符串，并返回新的字符串
<code>+split(regex: String): String[]</code>	返回一个字符串数组，包含被匹配模式的分割符分割的子字符串
<code>+split(regex: String, limit: int): String[]</code>	与前面的分割方法等同，除开 limit 参数控制了模式应用的次数

图 H-1 String 类包含使用正则表达式来匹配、替换和分割字符串的方法

显示

```
Jawi Jawi Jawi
```

下面代码

```
System.out.println("Java Java Java".replaceFirst("\\w", "wi"));
```

显示

```
Jawi Java Java
```

有两个重载的 split 方法。split(regex) 方法使用匹配的分割符将一个字符串分割为子字符串。例如，以下语句

```
String[] tokens = "Java1HTML2Perl".split("\\d");
```

将字符串 "Java1HTML2Perl" 分割为 Java、HTML 以及 Perl 并且保存在 tokens[0]，tokens[1] 以及 tokens[2] 中。

在 split(regex, limit) 方法中，limit 参数确定模式匹配多少次。如果 limit ≤ 0，split(regex, limit) 等同于 split(regex)。如果 limit > 0，模式最多匹配 limit - 1 次。下面是一些示例：

```
"Java1HTML2Perl".split("\\d", 0); 分割为 Java, HTML, Perl
"Java1HTML2Perl".split("\\d", 1); 分割为 Java1HTML2Perl
"Java1HTML2Perl".split("\\d", 2); 分割为 Java, HTML2Perl
"Java1HTML2Perl".split("\\d", 3); 分割为 Java, HTML, Perl
"Java1HTML2Perl".split("\\d", 4); 分割为 Java, HTML, Perl
"Java1HTML2Perl".split("\\d", 5); 分割为 Java, HTML, Perl
```

注意：默认的，所有的量词符都是“贪婪”的。这意味着它们会尽量匹配可能的最多次。比如，下面语句显示 JRvaa。因为第一个匹配成功的是 aaa。

```
System.out.println("Jaaavaa".replaceFirst("a+", "R"));
```

可以通过在后面添加问号符号来改变量词符的默认行为。量词符变为“不情愿”的，这意味着它将匹配尽可能少的次数。例如，下面的语句显示 JRaavaa，因为第一个匹配成功的是 a。

```
System.out.println("Jaaavaa".replaceFirst("a?", "R"));
```

枚举类型

1.1 简单枚举类型

枚举类型定义了一个枚举值的列表。每个值是一个标识符。例如，下面的语句声明了一个枚举类型，命名为 `MyFavoriteColor`，具有 `RED`、`BLUE`、`GREEN`、`YELLOW` 值。

```
enum MyFavoriteColor {RED, BLUE, GREEN, YELLOW};
```

枚举类型的值类似于一个常量，因此，按惯例拼写都是使用大写字母。因此，前面的声明采用 `RED`，而不是 `red`。按惯例，枚举类型命名类似于一个类，每个单词的第一个字母大写。

一旦定义了类型，就可以声明这个类型的变量了：

```
MyFavoriteColor color;
```

变量 `color` 可以具有定义在枚举类型 `MyFavoriteColor` 中的一个值，或者 `null`，但是不能具有其他值。Java 的枚举类型是类型安全的，这意味着试图赋一个枚举类型所列出的值或者 `null` 之外的一个值，都将导致编译错误。

枚举值可以使用下面的语法进行访问：

```
EnumeratedTypeName.valueName
```

例如，下面的语句将枚举值 `BLUE` 赋值给变量 `color`：

```
color = MyFavoriteColor.BLUE;
```

[1] 注意：必须使用枚举类型名称作为限定词来引用一个值，比如 `BLUE`。

如同其他类型一样，可以在一行语句中来声明和初始化一个变量：

```
MyFavoriteColor color = MyFavoriteColor.BLUE;
```

枚举类型被作为一个特殊的类来对待。因此，枚举类型的变量是引用变量。一个枚举类型是 `Object` 类和 `Comparable` 接口的子类。因此，枚举类型继承了 `Object` 类中的所有方法，以及 `Comparable` 接口中的 `compareTo` 方法。另外，可以在一个枚举类型的对象上面使用下面的方法：

- `public String name();`

为对象返回名字值。

- `public int ordinal();`

返回和枚举值关联的序号值。枚举类型中的第一个值具有序号数 0，第二个值具有序号值 1，第三个为 2，依次类推。

程序清单 I-1 给出了一个程序，展示了枚举类型的使用。

程序清单 I-1 EnumeratedTypeDemo.java

```

1 public class EnumeratedTypeDemo {
2     static enum Day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
3         FRIDAY, SATURDAY};
4
5     public static void main(String[] args) {
6         Day day1 = Day.FRIDAY;
7         Day day2 = Day.THURSDAY;
8
9         System.out.println("day1's name is " + day1.name());
10        System.out.println("day2's name is " + day2.name());
11        System.out.println("day1's ordinal is " + day1.ordinal());
12        System.out.println("day2's ordinal is " + day2.ordinal());
13
14        System.out.println("day1.equals(day2) returns " +
15            day1.equals(day2));
16        System.out.println("day1.toString() returns " +
17            day1.toString());
18        System.out.println("day1.compareTo(day2) returns " +
19            day1.compareTo(day2));
20    }
21 }

```

```

day1's name is FRIDAY
day2's name is THURSDAY
day1's ordinal is 5
day2's ordinal is 4
day1.equals(day2) returns false
day1.toString() returns FRIDAY
day1.compareTo(day2) returns 1

```

在第 2 ~ 3 行定义了枚举类型 Day。变量 day1 和 day2 声明为 Day 类型，在第 6 ~ 7 行赋枚举值。由于 day1 的值为 FRIDAY，它的序号值为 5（第 11 行）。由于 day2 的值为 THURSDAY，它的序号值为 4（第 12 行）。

由于一个枚举类型是 Object 类和 Comparable 接口的子类。可以从一个枚举对象引用变量调用 equals、toString 以及 compareTo 方法（第 14 ~ 19 行）。如果 day1 和 day2 具有同样的序号数，day1.equals(day2) 返回真。day1.compareTo(day2) 返回 day1 的序号数到 day2 的序号数之间的差距。

作为另外一种选择，可以将程序清单 I-1 中的代码重新写为程序清单 I-2。

程序清单 I-2 StandaloneEnumTypeDemo.java

```

1 public class StandaloneEnumTypeDemo {
2     public static void main(String[] args) {
3         Day day1 = Day.FRIDAY;
4         Day day2 = Day.THURSDAY;
5
6         System.out.println("day1's name is " + day1.name());
7         System.out.println("day2's name is " + day2.name());
8         System.out.println("day1's ordinal is " + day1.ordinal());
9         System.out.println("day2's ordinal is " + day2.ordinal());
10
11        System.out.println("day1.equals(day2) returns " +
12            day1.equals(day2));
13        System.out.println("day1.toString() returns " +
14            day1.toString());
15        System.out.println("day1.compareTo(day2) returns " +
16            day1.compareTo(day2));
17    }

```

```
18 }  
19  
20 enum Day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,  
21 FRIDAY, SATURDAY}
```

枚举类型可以在一个类中定义，如程序清单 I-1 中的第 2～3 行所示；或者单独定义，如程序清单 I-2 的第 20～21 行所示。在前一种情况下，枚举类型被作为内部类对待。程序编译后，将创建一个名为 `EnumeratedTypeDemo$Day` 的类。在后一种情况下，枚举类型作为一个独立的类来对待。程序编译后，将创建一个名为 `Day.class` 的类。

注意：当一个枚举类型在一个类中声明时，类型必须声明为类的一个成员，而不能在一个方法中声明。而且，类型总是 `static` 的。由于这个原因，程序清单 I-1 第 2 行的 `static` 关键字可以省略。可以用于内部类的可见性修饰符也可以应用到在一个类中定义的枚举类型中。

技巧：使用枚举值（例如，`Day.MONDAY`，`Day.TUESDAY`，等等）而不是字面量整数值（例如，0，1，等等）可以让程序更加易于阅读和维护。

1.2 通过枚举变量使用 if 或者 switch 语句

枚举变量具有一个值。程序经常需要根据取值来执行特定的动作。例如，如果值为 `Day.MONDAY`，则踢足球；如果值为 `Day.TUESDAY`，则学习钢琴课，等等。可以使用 `if` 语句或者 `switch` 语句来测试变量的值，如图 a) 和 b) 所示。

```
if (day.equals(Day.MONDAY)) {  
    // process Monday  
}  
else if (day.equals(Day.TUESDAY)) {  
    // process Tuesday  
}  
else  
    ...
```

a)

等价于

```
switch (day) {  
    case MONDAY:  
        // process Monday  
        break;  
    case TUESDAY:  
        // process Tuesday  
        break;  
    ...  
}
```

b)

在 b 图的 `switch` 语句中，`case` 标签是一个无限定词的枚举值（即，`MONDAY`，而不是 `Day.MONDAY`）。

1.3 使用 foreach 循环处理枚举值

每个枚举类型有一个静态方法 `values()`，可以返回这个类型中所有的枚举值到一个数组中。例如，

```
Day[] days = Day.values();
```

可以使用通常的循环如图 a 中所示，或者图 b 中的 `foreach` 循环来处理数组中的所有值。

```
for (int i = 0; i < days.length; i++)  
    System.out.println(days[i]);
```

a)

等价于

```
for (Day day: days)  
    System.out.println(day);
```

b)

1.4 具有数据域，构造方法和方法的枚举类型

前面介绍的简单枚举类型定义了一个类型，具有一个枚举值的列表。也可以定义一个具

有数据域，构造方法和方法的枚举类型，如程序清单 I-3 所示。

程序清单 I-3 TrafficLight.java

```
1 public enum TrafficLight {  
2     RED ("Please stop"), GREEN ("Please go"),  
3     YELLOW ("Please caution");  
4  
5     private String description;  
6  
7     private TrafficLight(String description) {  
8         this.description = description;  
9     }  
10  
11     public String getDescription() {  
12         return description;  
13     }  
14 }
```

第 2~3 行定义了枚举值。值的声明必须是类型声明的第一条语句。一个名为 `description` 的数据域在第 5 行声明，描述了一个枚举值。构造方法 `TrafficLight` 在第 7~9 行声明。当访问枚举值的时候，构造方法将被调用。枚举值的参数将传递给构造方法，在构造方法中赋值给 `description`。

程序清单 I-4 给出了一个使用 `TrafficLight` 的测试程序。

程序清单 I-4 TestTrafficLight.java

```
1 public class TestTrafficLight {  
2     public static void main(String[] args) {  
3         TrafficLight light = TrafficLight.RED;  
4         System.out.println(light.getDescription());  
5     }  
6 }
```

一个枚举值 `TrafficLight.RED` 赋值给变量 `light` (第 3 行)。访问 `TrafficLight.RED` 引起 JVM 使用参数 “please stop” 调用构造方法。枚举类型中的方法是和类中的方法调用一样的。`light.getDescription()` 返回对枚举值的描述 (第 4 行)。

【注意】 Java 语法要求枚举类型的构造方法是私有的，避免被直接调用。私有修饰符可以省略。在这种情况下，被默认为是私有的。

推荐阅读



中文版
第2版

作者: Mark Allen Weiss 著
书号: 7-111-12748-X, 35.00元



中文版
第2版

作者: Mark Allen Weiss 著
书号: 978-7-111-23183-7, 55.00元
第3版中文版即将在2016年出版



中文版
第2版

作者: Sartaj Sahni 著
书号: 978-7-111-49600-7, 79.00元



中文版
第2版

作者: Randal E. Bryant 等著
书号: 978-7-111-32133-0, 99.00元



中文版
第5版

作者: David A. Patterson John L. Hennessy
中文版: 978-7-111-50482-5, 99.00元



中文版
第6版

作者: James F. Kurose 等著
书号: 978-7-111-45378-9, 79.00元



中文版
第6版

作者: Abraham Silberschatz 著
中文翻译版: 978-7-111-37529-6, 99.00元
本科教学版: 978-7-111-40085-1, 59.00元



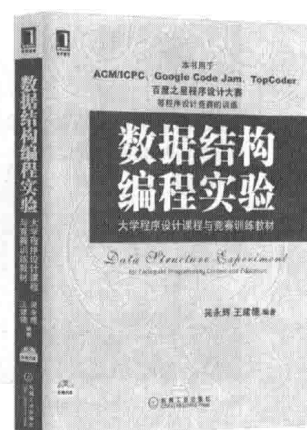
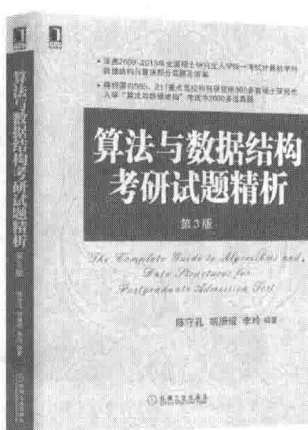
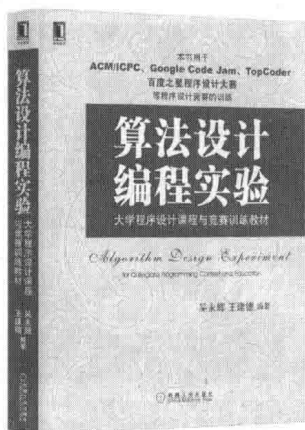
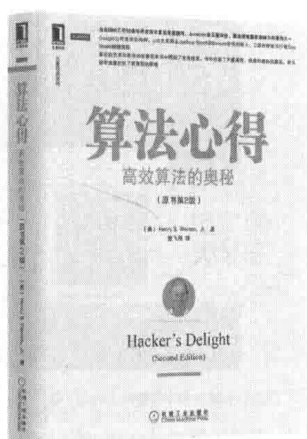
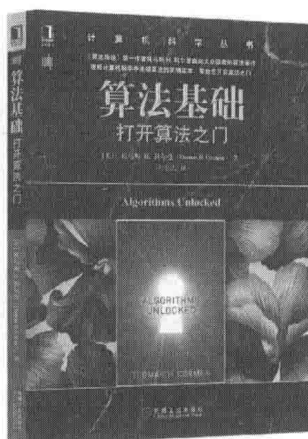
中文版
第3版

作者: Jiawei Han 等著
中文版: 978-7-111-39140-1, 79.00元



作者: Thomas Erl 等著
中文版: 978-7-111-46134-0, 69.00元

推荐阅读



算法导论（原书第3版）

作者：Thomas H.Cormen 等 ISBN：978-7-111-40701-0 定价：128.00元

算法基础：打开算法之门

作者：Thomas H. Cormen ISBN：978-7-111-52076-4 定价：59.00元

算法心得：高效算法的奥秘（原书第2版）

作者：Henry S. Warren ISBN：978-7-111-45356-7 定价：89.00元

算法设计编程实验：大学程序设计课程与竞赛训练教材

作者：吴永辉 ISBN：978-7-111-42383-6 定价：69.00元

算法与数据结构考研试题精析 第3版

作者：陈守孔 ISBN：978-7-111-50067-4 定价：69.00元

数据结构编程实验：大学程序设计课程与竞赛训练教材

作者：吴永辉 ISBN：978-7-111-37395-7 定价：59.00元

Java语言程序设计（进阶篇）原书

Introduction to Java Programming Comprehensive V

本书是Java语言的经典教材，多年来畅销不衰。本书全面整合了Java 8的特性，采用“基础优先，问题驱动”的教学方式，循序渐进地介绍了程序设计基础、解决问题的方法、面向对象程序设计、图形用户界面设计、异常处理、I/O和递归等内容。此外，本书还全面且深入地覆盖了一些高级主题，包括算法和数据结构、多线程、网络、国际化、高级GUI等内容。

本书中文版由《Java语言程序设计 基础篇》和《Java语言程序设计 进阶篇》组成。基础篇对应原书的第1~18章，进阶篇对应原书的第19~33章。为满足对Web设计有浓厚兴趣的同学，本版在配套网站上增加了第34~42章的内容，以提供更多的相关信息。

本书特点

- 基础篇介绍基础内容，进阶篇介绍高级内容，便于教师按需选择理想的教材。
- 全面整合了Java 8的特性，对全书的内容进行了修订和更新，以反映Java程序设计的最新技术发展。
- 对面向对象程序设计进行了深入论述，包含GUI程序设计的基础和扩展。
- 大量示例中都包括问题求解的详细步骤，很多示例都是随着Java技术的引入而不断进行增强，更易于学习。
- 用JavaFX取代了Swing，极大地简化了GUI编程。
- 通过更多有趣的示例和练习激发学生的兴趣，并在配套网站上额外为教师提供了100多道编程练习题。

作者简介

梁勇 (Y. Daniel Liang) 现为阿姆斯特朗亚特兰大州立大学计算机科学系教授。之前曾是普渡大学计算机科学系副教授，并两次获得普渡大学杰出研究奖。他所编写的Java教程在美国大学Java课程中采用率极高，同时他还兼任Prentice Hall Java系列丛书的编辑。他是“Java Champion”荣誉得主，并在世界各地为在校学生和程序员做Java程序设计方法及技术方面的讲座。

译者简介

戴开宇 复旦大学软件学院教师，工程硕士导师，中国计算机学会会员。博士毕业于上海交通大学计算机应用专业，2011~2012年在美国佛罗里达大学作访问学者。承担多门本科专业课程、通识教育课程以及工程硕士课程，这些课程被评为校精品课程、上海市重点建设课程、IBM-教育部精品课程等。



www.pearson.com



投稿热线: (010) 88379604
客服热线: (010) 88378991 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

封面设计: 包易 林杉

上架指导: 计算机\程序设计

ISBN 978-7-111-54856-0



9 787111 548560 >

定价: 89.00元